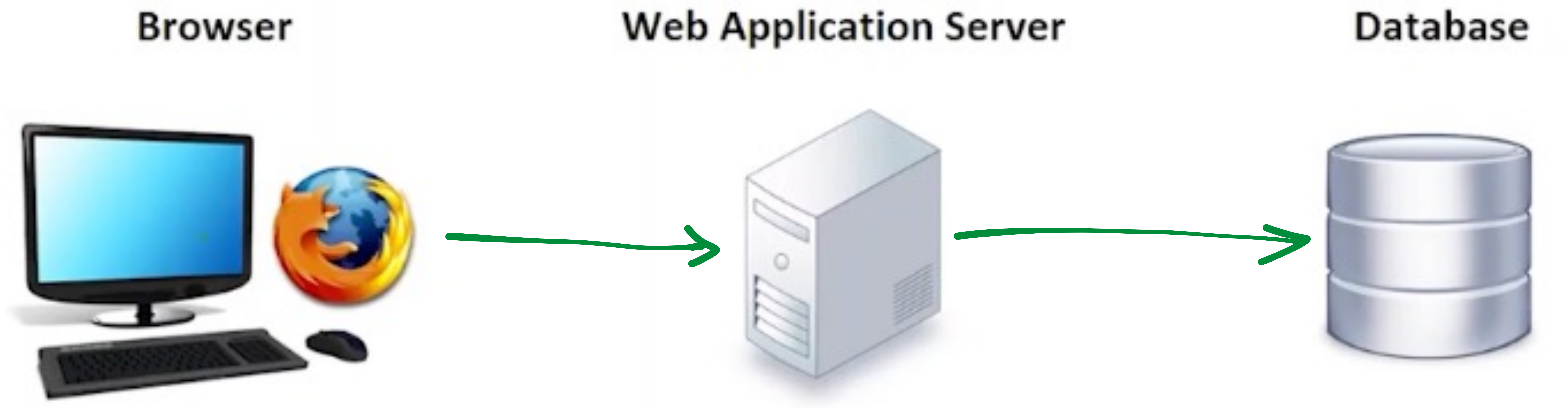
Web Security

SQL Injection

Web Application Architecture



What is SQL?

- **S**tructured **Q**uery **L**anguage
- A computer language for **storing**, **manipulating** and **retrieving** data stored in a relational database

Getting Started With Select

- A SQL database contains a bunch of tables

sales

client	item
〰	〰
〰	〰
〰	〰
〰	〰
〰	〰
〰	〰

clients

id	name
〰	〰
〰	〰
〰	〰
〰	〰
〰	〰
〰	〰

cats

owner	name
〰	〰
〰	〰
〰	〰
〰	〰
〰	〰
〰	〰

Getting Started With Select

- Every SELECT query takes data from those tables and outputs the results

cats

owner	name
1	cheddar
1	daisy
3	buttercup
4	fluffy
4	zeus
5	ruby

query

```
SELECT *
FROM cats
WHERE owner = 1
```

query output

owner	name
1	cheddar
1	daisy

Getting Started With Select

- Every SELECT query takes data from those tables and outputs the results

cats

owner	name
1	cheddar
1	daisy
3	buttercup
4	fluffy
4	zeus
5	ruby

query

```
SELECT name
FROM cats
WHERE owner = 1
```

query output

name
cheddar
daisy

Update a Row

- We can use UPDATE to modify an existing row in a table

cats

owner	name
1	cheddar
1	daisy
3	buttercup
4	fluffy
4	zeus
5	ruby

query

```
UPDATE cats
SET name = ‘paws’
WHERE owner = 3
```

cats

owner	name
1	cheddar
1	daisy
3	paws
4	fluffy
4	zeus
5	ruby

Insert a Row

- We can use INSERT INTO to insert a new row into a table

cats

owner	name
1	cheddar
1	daisy
3	paws
4	fluffy
4	zeus
5	ruby

query

```
INSERT INTO cats
VALUES (6, boots)
```

cats

owner	name
1	cheddar
1	daisy
3	paws
4	fluffy
4	zeus
5	ruby
6	boots

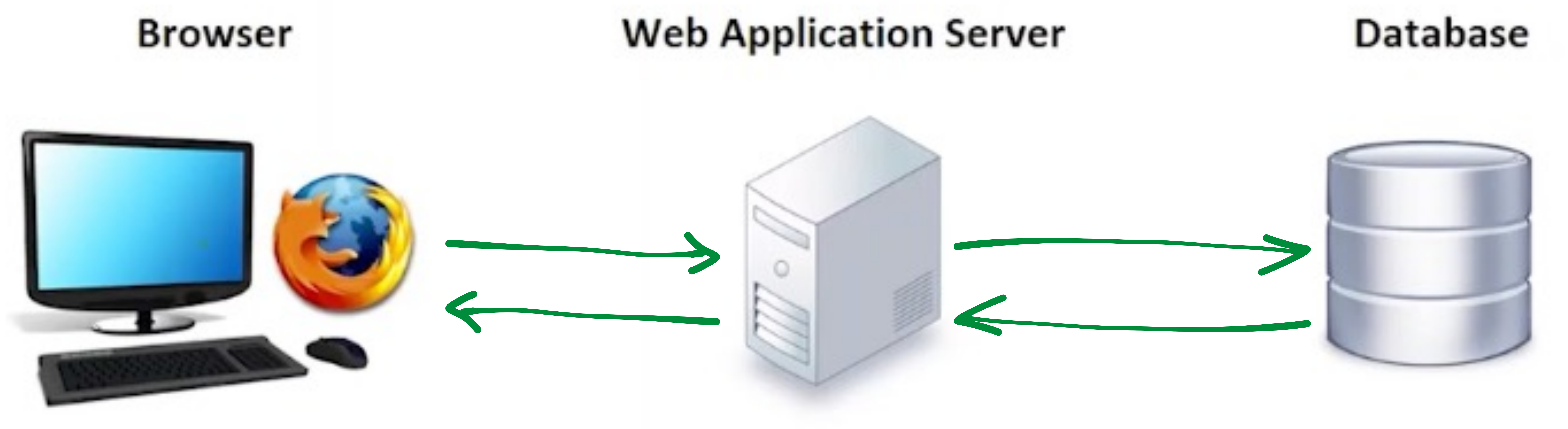
SQL Injection

SQL injection is a code injection attack on data-driven applications, in which malicious SQL statements are inserted into an entry field for execution

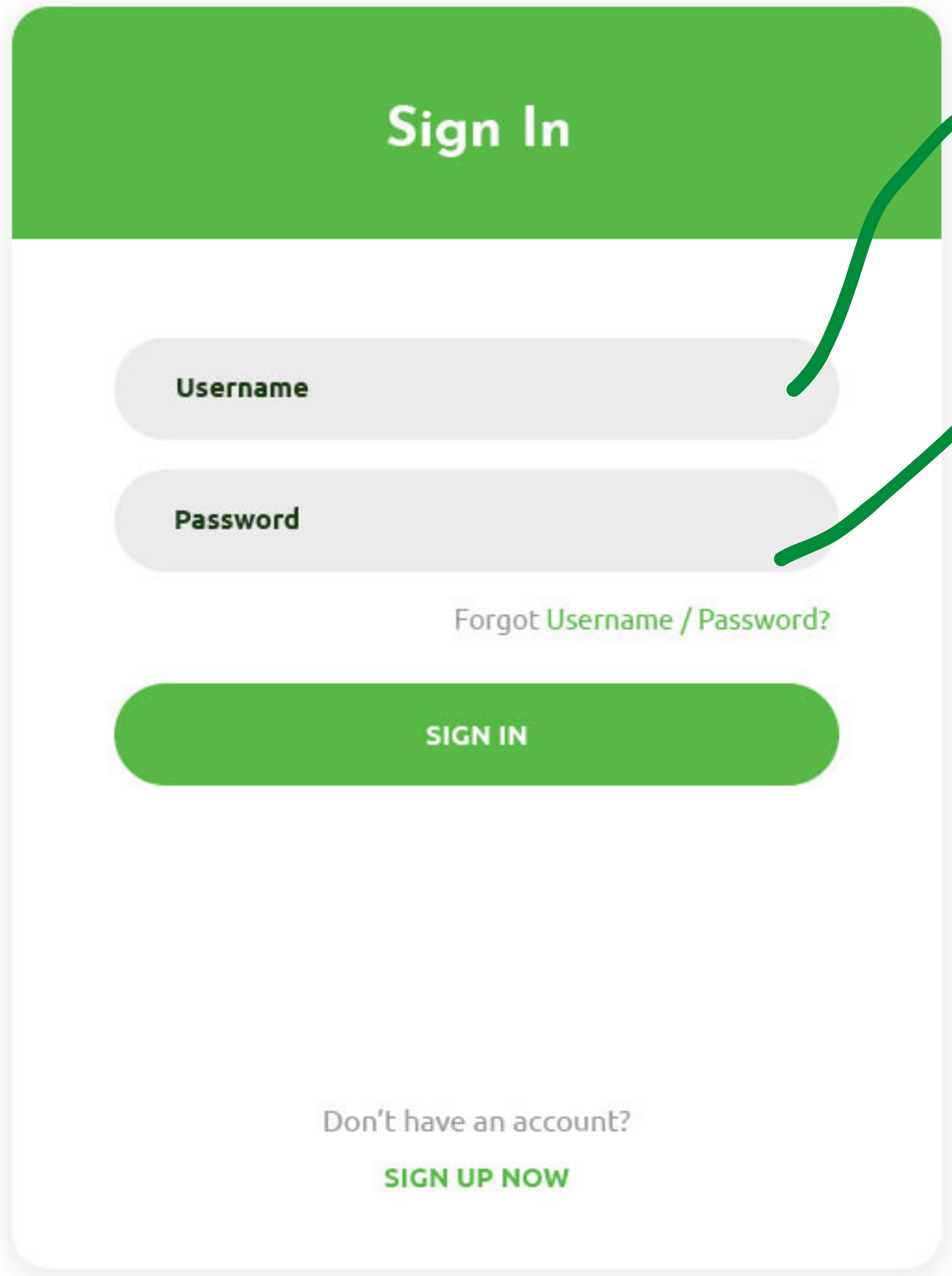
<u>Goals</u>: Change or exfiltrate info from a database

Main idea: Inject code through the parts of a query that you define

Web Application Architecture



SQL Injection Example



```
$login = $_POST['login'];
$pass = $_POST['password'];

$sql = "SELECT id FROM users
        WHERE username = '$login'
        AND password = '$password'";

$rows = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Non-Malicious Input

```
$login = $_POST['login’]; // dbalash
$pass = $_POST['password']; // P@ssw0rd123!


$sql = "SELECT id FROM users WHERE uid = ‘$login' AND pwd = ‘$pass'”;


$rows = $db->executeQuery($sql);

if $rows.count > 0 {
    // Success!
}
```

Non-Malicious Input

```
$login = $_POST['login’]; // dbalash
$pass = $_POST['password']; // P@ssw0rd123!

                                            dbalash               P@ssw0rd123!
$sql = "SELECT id FROM users WHERE uid = ‘$login' AND pwd = ‘$pass'”;


$rows = $db->executeQuery($sql);

if $rows.count > 0 {
   // Success!
}
```

Bad Input

```
$login = $_POST['login’]; // dbalash
$pass = $_POST['password']; // P@ssw0rd123!'

$sql = "SELECT id FROM users WHERE uid = ‘$login' AND pwd = ‘$pass'”;

$rows = $db->executeQuery($sql);     // SQL Syntax Error
```

Malicious Input

```
$login = $_POST['login']; // dbalash'--
$pass = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = ‘$login' AND pwd = ‘$pass'”;
//      "SELECT id FROM users WHERE uid = ‘dbalash'-- COMMENTED OUT

$rows = $db->executeQuery($sql); //(No Error)

if $rows.count > 0 {
   // Success!
}
```

No Username Needed!

```
$login = $_POST['login’]; // 'or 1=1 --
$pass = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = ‘$login' AND pwd = ‘$pass'”;
//      "SELECT id FROM users WHERE uid = ''or 1=1 -- COMMENTED OUT

$rows = $db->executeQuery($sql); // (No Error)

if $rows.count > 0 {
   // Success!
}
```

Causing Damage

```
$login = $_POST[‘login’]; // '; DROP TABLE [users] --
$pass = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = ‘$login' AND pwd = ‘$pass'”;
//      "SELECT id FROM users WHERE uid = ’’; DROP TABLE [users] --

$rows = $db->executeQuery($sql);
// No Error…(and no more users table)
```

Preventing SQL Injection

Never trust user input (*particularly* when constructing a command)

Never manually build SQL commands yourself!

Sanitize / Escape user input (like XSS, this is harder than you think!)

There are tools for safely passing user input to databases:

- Parameterized (AKA Prepared) SQL
- ORM (Object Relational Mapper) -> uses Prepared SQL internally

Parameterized SQL

Parameterized SQL allows you to send query and arguments separately to server

```
sql = “INSERT INTO users(name, email) VALUES(?,?)”
cursor.execute(sql, [David Balash', ‘dbalash@richmond.edu'])

sql = "SELECT * FROM users WHERE email = ?"
cursor.execute(sql, [‘dbalash@richmond.edu'])
```

Values are sent to server separately from command. Library doesn’t need to escape

Benefit 1: No need to escape untrusted data — server handles behind the scenes

Benefit 2: Parameterized queries are *faster* because server caches query plan

Object Relational Mappers

Object Relational Mappers (ORM) provide an interface between native objects and relational databases.

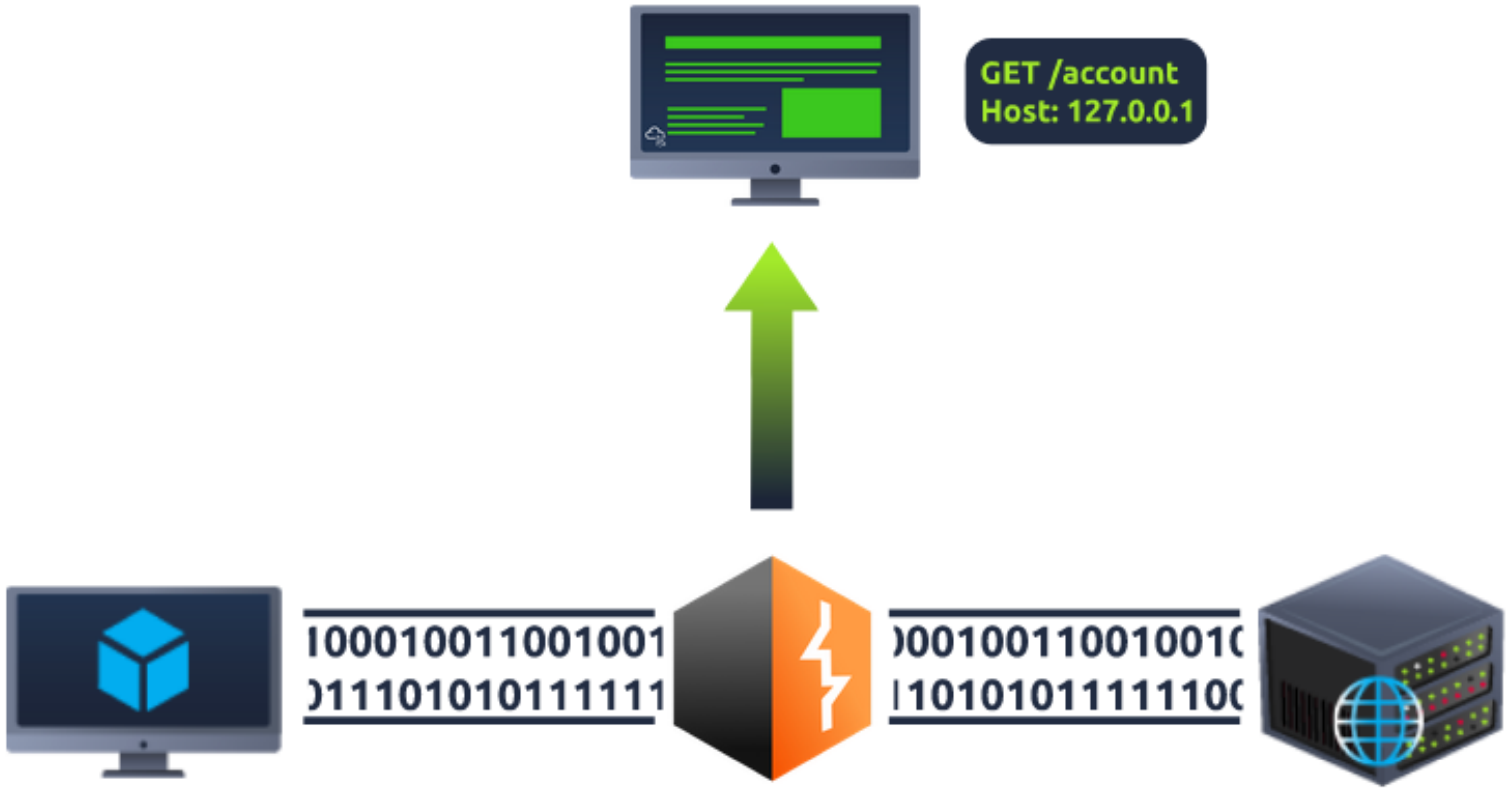
```
class User(DBObject):

    __id__  = Column(Integer, primary_key=True)
    name    = Column(String(255))
    email   = Column(String(255), unique=True)

if __name__ == "__main__":
    users = User.query(email=dbalash@richmond.edu').all()
```

Burp Suite

Burp Suite captures and enables manipulation of all the HTTP/HTTPS traffic between a browser and a web server


GET /account
Host: 127.0.0.1
1000100110010011
0111010101111111
0001001100100110
1101010111111100