Using a Safe Language

Enough with the buffer overflows!

Let's use a **safe** language, where overflows are not possible*

*as long as there are no bugs in the language† implementation

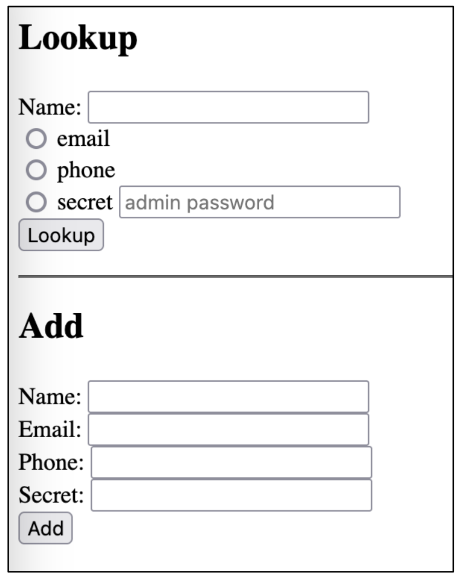
† or libraries that the language uses

Safe means no undefined behavior, so you can reason in terms of source without considering generated machine code

A safe language can be misused so that it allows a **code injection** attack

A JavaScript-Based Server

See `server.js`



Lookup via Eval

```
function lookup(name, info) {
    try {
        // Use `eval` to get any field. How convenient!
        return eval("directory['" + name + "']." + info);
    } catch (error) {
        console.log(error);
        return "lookup failed";
    }
}
```

Lookup via Eval

```
function lookup(name, info) {
    try {
        // Use `eval` to get any field. How convenient!
        return eval("directory['" + name + "'']." + info);
    } catch (error) {
        console.log(error);
        return "lookup failed";
    }
}
```

No gurantee that input is just a field name

This implementation is susceptible to a **code injection** attack

```
curl "http://localhost:8085/?name=Alice&info=info%2Badmin_password&lookup=Lookup"
```

The first rule of using `eval`: *do not use `eval`*

Lookup via Dictionary Key

```
function lookup(name, info) {
    try {
        return directory[name][info];
    } catch (error) {
        console.log(error);
        return "lookup failed";
    }
}
```

Lookup via Dictionary Key

```
function lookup(name, info) {
    try {
        return directory[name][info];
    } catch (error) {
        console.log(error);
        return "lookup failed";
    }
}
```

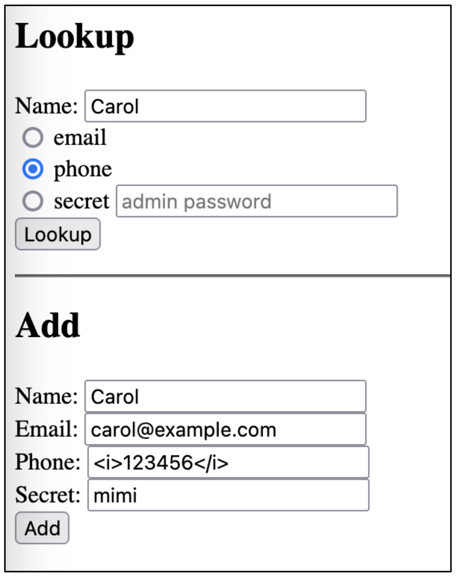
Still could be something like "__proto__" to access object internals

All inputs need to be **validated** and/or **sanitized** on the server side

Lookup with Validated Key

```
function lookup(name, info) {
    try {
        if (is_valid_field_name(info))
            return directory[name][info];
        else
            throw new Error("invalid field")
    } catch (error) {
        console.log(error);
        return "lookup failed";
    }
}
```

HTML Injection



123456

```
res.end("<html>"
        + "<body>"
        + info
        + "</body>"
        + "</html>");
```

Unsanitized input as unescaped output creates the possibility of an **HTML injection** attack

The injected HTML could include JavaScript code that runs in the browser of anyone who looks for Carol's phone number

Validation

Client-side validation

e.g., using JavaScript to check text before sending

An important component of a *good user interface*

Server-side validation

e.g., `is_valid_field` and `sanitize`

An import component of **security**

Code at Run Time

Using `eval` is usually just bad, but the case of HTML illustrates how run-time generation of *some* code is often necessary

Another common code-generation case: accessing a database

Databases

A **relational database** keeps data in tables

name	*email*	*phone*	*secret*
Alice	alice@example.com	801-555-1212	clock tower
Bob	bob@example.com	385-555-1212	house
...			

To handle large amounts of data, there are many special implementation techniques for representing and managing theses tables

Database Operations

The suite of operations on a database is known as **CRUD**:

Create: create a table or new rows in a table
Read: find data in tables
Update: change data in tables
Delete: remove a table or rows from a table

Combinations of operations can be complex, so to encode CRUD requests, there's a whole language: **Structured Query Language (SQL)**

SQL Examples: Managing Tables

```
CREATE TABLE people (name TEXT,
                     email TEXT,
                     phone TEXT,
                     secret TEXT)

DROP TABLE people
```

SQL Examples: Querying Tables

```
SELECT phone FROM people WHERE name='Alice'

SELECT * FROM people WHERE name='Alice' OR secret='house'
```

SQL Examples: Updating Tables

```
INSERT INTO people VALUES ('Alice',
                           'alice@example.com',
                           '801-555-1212',
                           'clock tower')

DELETE FROM people WHERE name='Bob'
```

A JavaScript Server using a Database

See `server_db.js`

SQL Injection Attacks

Part of the problem here is using a multi-statement `exec` where a single-statement `run` is better

Building SQL Commands

Instead of building a string with values, use substitutions as supported by a SQL binding

```
db.run("INSERT INTO people VALUES (?, ?, ?, ?)",
       [name, email, phone, secret])


if (is_valid_field_name(info)) {
    db.get("SELECT " + info + " FROM people WHERE name=?", [ name ],
           ....)
}
```

JSON in JavaScript

JSON is a format for data exchange that is often used in JavaScript systems

```
{ "name": "Alice", "number": "801-555-1212" }
```

Since JSON looks like JavaScript, it suggests

```
function parse_json(str) {
   return eval("(" + str + ")");
}
```

Obviously, don't do that, and instead use `JSON.parse`, which is roughly

```
function parse(str) {
    validate_json(str);
    return eval("(" + str + ")");
}
```

Summary

Code-injection attacks are not just for unsafe languages

- **HTML injection** vulnerabilities remain common
- **SQL injection** vulnerabilities remain common

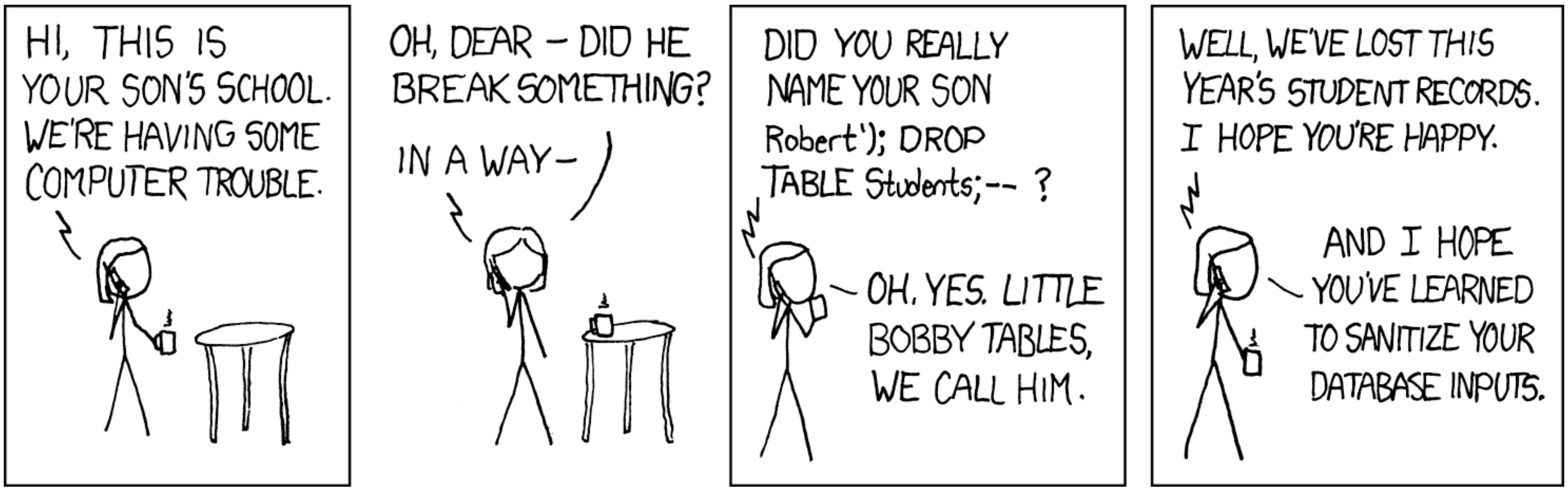
Try a web search for “SQL injection news”

The general problem is that code is data, and data can become code

Be alert for contexts that use data as code:

- When possible, switch to an API that doesn’t do that
- Take care parsing when **validating** and/or **sanitizing** input
- Take care printing when **sanitizing** and/or **escaping** output

Little Bobby Tables



https://xkcd.com/327/