CS 152: *Programming Language Paradigms*

Dynamic Code Evaluation & Taint Analysis

Prof. Tom Austin

San José State University

Dynamic code evaluation

`eval`

- Executes dynamically
- Typically, `eval` takes a string:
  `eval "puts 2+3"`
- Popular feature
  - especially in JavaScript
    - Richards et al. *The Eval that Men Do*, 2011
- Source of security problems

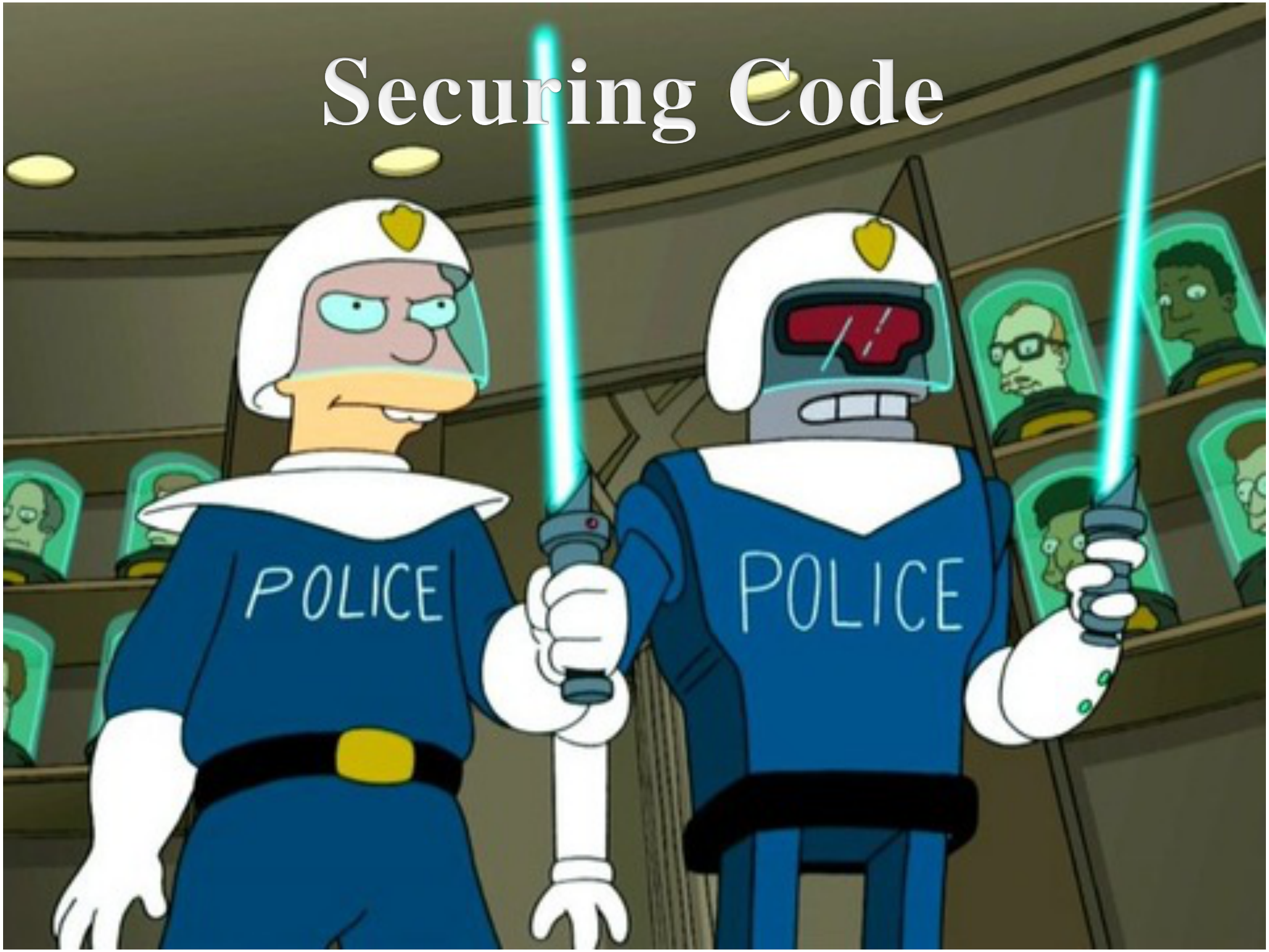
Parsing JSON
(in-class)

Review: additional Ruby `eval` methods

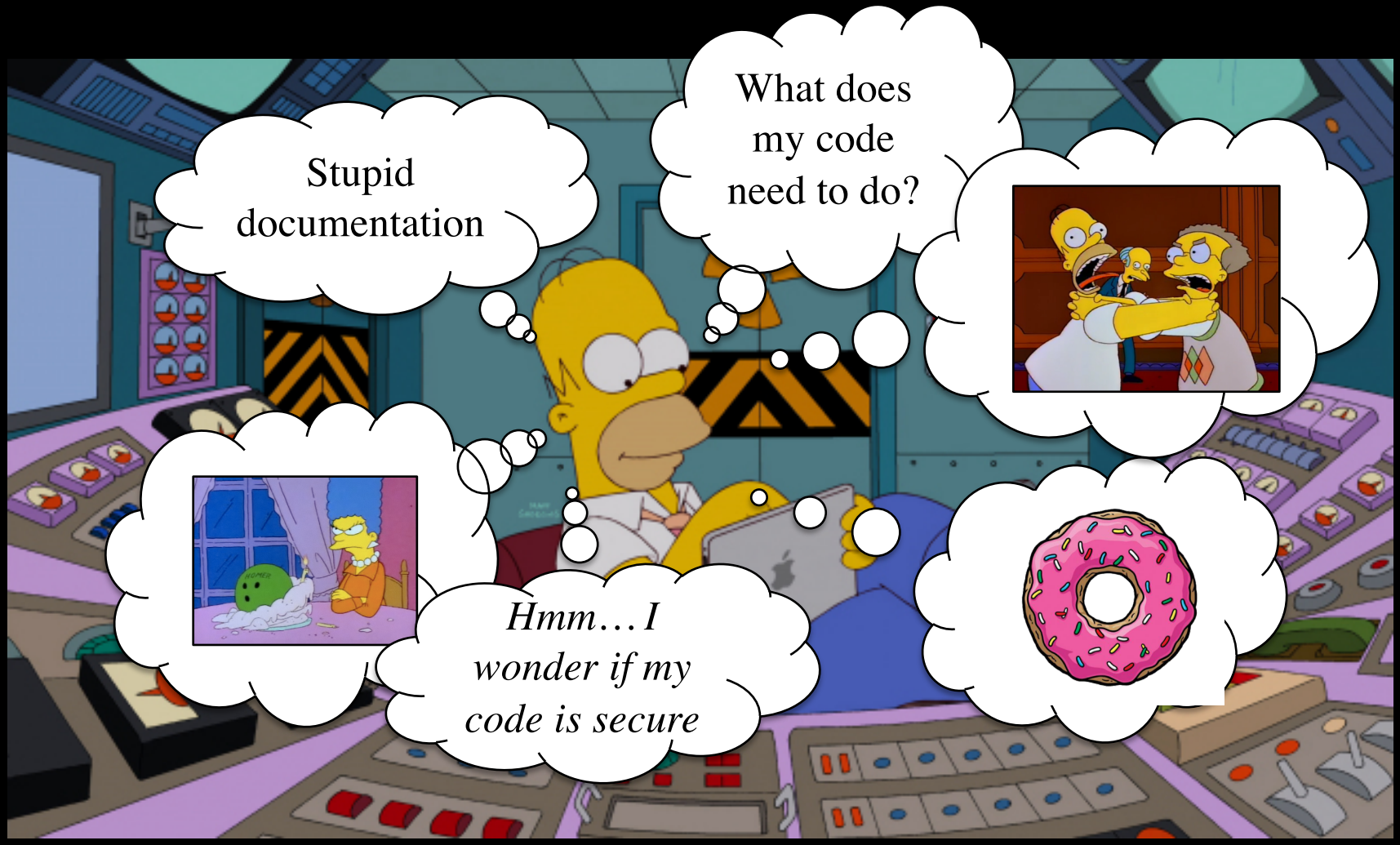
- `instance_eval` evaluates code within the body of an object.
- `class_eval` evaluates code within the body of a class.
- These methods can take a string or (more safely) a block of code.

`class_eval` example

(in class)


Securing Code
POLICE
POLICE

The mind of a developer



Web Security in the News

How do companies/developers cope?

- Train/shame developers to follow best practices.
- Hire security experts
- Use analysis tools
- Hush up mistakes
- Budget to handle emergencies
- Bury their heads in the sand.

Secure By Architecture

Developers make mistakes.

Can we design tools to create secure systems, despite developer mistakes?

Success story: memory-safe languages

- Buffer overflows were once ubiquitous
- Memory-safe languages manage memory automatically
  - Developer focus on functionality
  - Security-critical bugs are eliminated
- Buffer overflows have virtually disappeared
  - Except in your OS, web browser, etc.

Two Security Mechanisms

- **Taint analysis**:
  - protect critical fields from "dirty" data
- **Information flow analysis**:
  - Prevent secrets from leaking.

Taint Analysis: Protecting against dirty data

Taint analysis

- **Taint analysis** focuses on *integrity:*
  - does "dirty" data corrupt trusted data?
- Integrated into Perl and Ruby
- Handles *explicit flows* only
  - direct assignment
  - passing parameters

Attacks preventable by taint analysis

- Data under the control of the user may pose a security risk
  - SQL injection
  - cross-site scripting (XSS)
  - cross-site request forgery (CSRF)
- Taint tracking tracks untrusted variables and prevents then from being used in unsafe operations

Taint Tracking History

- 1989 – Perl 3 support for a taint mode
- 1996 – Netscape included support for a taint mode in server-side JavaScript
  - Later abandoned
- Ruby later implemented a taint mode; we'll review in more depth.

Taint Mode in Ruby

- Protect against integrity attacks.
  - E.g. Data pulled from an HTML form cannot be passed to `eval`.
- Cannot taint booleans or ints.
- Multiple ways to run in safe mode:
  - Use `-T` command line flag.
  - Include `$SAFE` variable in code.

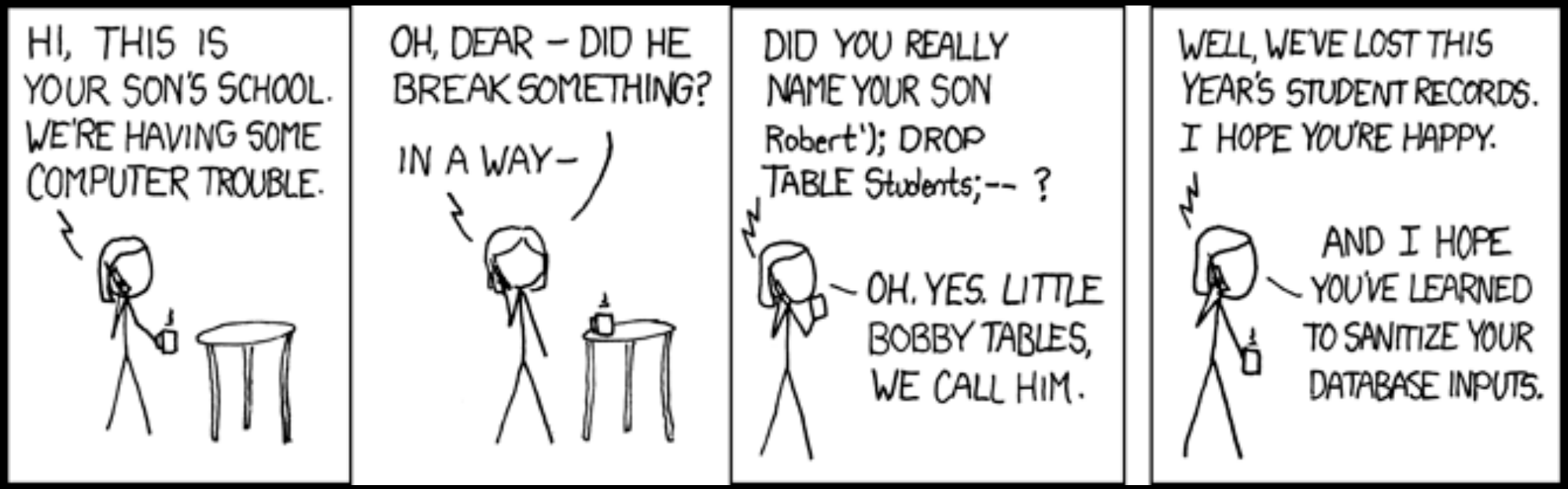
`$SAFE` levels in Ruby

- 0 – No checking (default)
- 1
  - Tainted data cannot be passed to eval
  - Cannot load/require new files
- 2 – Can't change, make, or remove directories
- 3
  - New strings/objects are automatically tainted
  - Cannot untaint tainted values
- 4 – Safe objects become immutable

```
s = "puts 4-3".taint
$SAFE = 1 # Can't eval tainted data
s.untaint # Removes taint from data
puts s.tainted?
eval s

$SAFE = 3
s2 = "puts 2 * 7" # Tainted
s2.untaint # Won't work now
eval s2
eval s # this is OK
```

SQL injection



"Exploits of a Mom", http://xkcd.com/327/

An improperly sanitized query

```
<?php
pg_query($con,
   "INSERT INTO STUDENTS (name)"
   . " VALUES('" . $_POST['name'].
   "')";
?>
```

name = "Joe"

```
INSERT INTO STUDENTS (name)
   VALUES('Joe')
```

An improperly sanitized query

```
<?php
pg_query($con,
   "INSERT INTO STUDENTS  (name)"
   . " VALUES('" . $_POST['name'].
   "')";
?>
```

```
name = "Robert');
DROP TABLE STUDENTS;--"
```

Quote ends data: now in SQL

```
INSERT INTO STUDENTS (name)
    VALUES('Robert');
DROP TABLE STUDENTS;--')
```

Remainder commented out.

Correctly sanitized query

```
                                name = "Robert');
<?php                           DROP TABLE STUDENTS;--"
pg_query_params($con,
   "INSERT INTO STUDENTS (name)"
   . " VALUES($1)",
   ARRAY($_POST['name']);
?>
```

Quote escaped.

```
INSERT INTO STUDENTS (name)
   VALUES('Robert');
DROP TABLE STUDENTS;--')
```

```
# Data from web
s = "Robert'); DROP TABLE " +
    "STUDENTS;--"
s.taint
exec_query("SELECT *" +
           " FROM STUDENTS" +
           " WHERE NAME='" +
           s + "';"
```

```
class Record
  def exec_query(query_str)
      if query_str.tainted?
      puts "Err: tainted string"
        else
            # Perform the query
            ...
    end
  end
end
```

Lab: Taint tracking

Today's lab explores taint tracking in Ruby.

Starter code is available on the course website.

Details in Canvas.