SQL and other injections

Secure Programming
Lecture 5

Where are we?

- We saw a process we can follow to help us find vulnerabilities in an application
- Now it's time to look into specific classes of vulnerabilities (and exploits)
- But before we start…

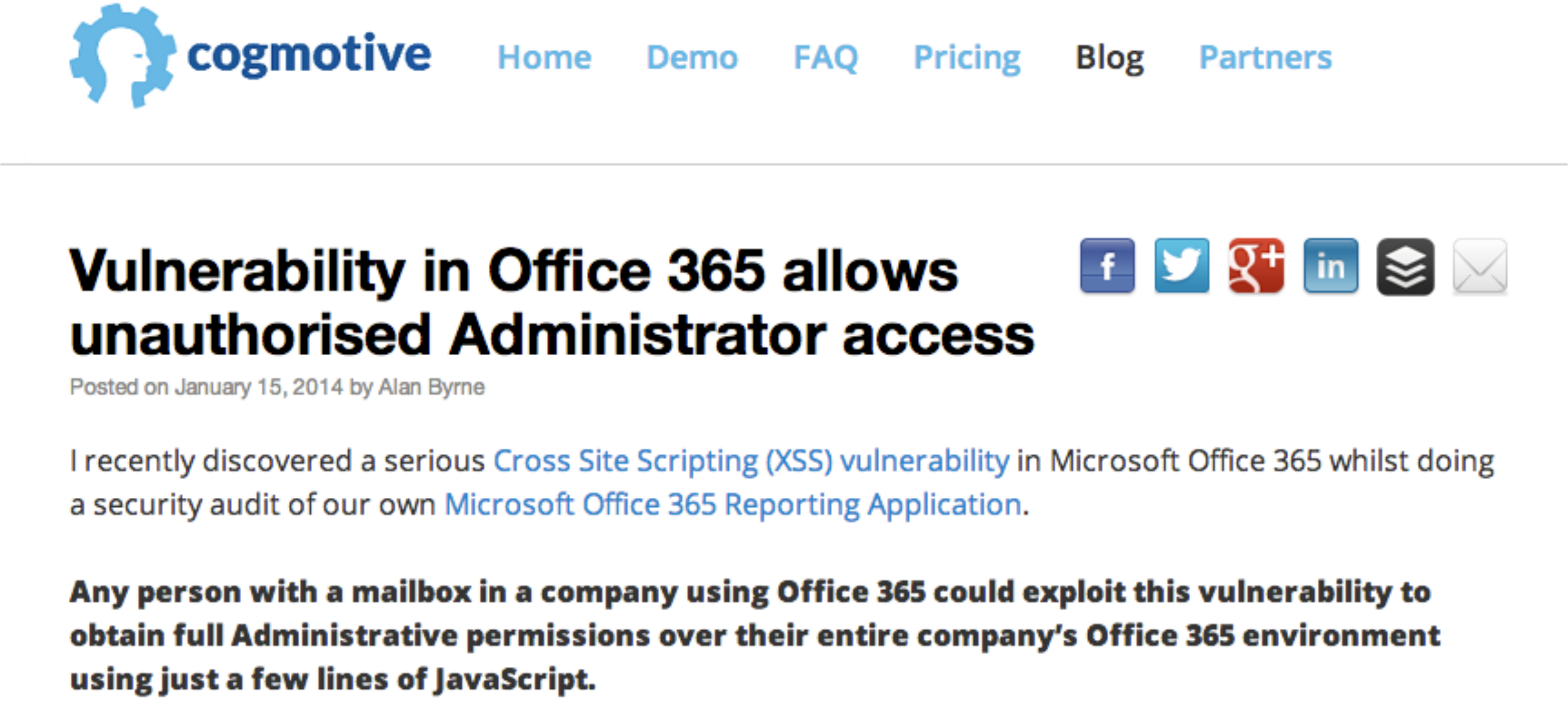
Review

Are you:

- Confused about the security principles, but
- A great fan of Star Wars?

This may help: http://emergentchaos.com/the-security-principles-of-saltzer-and-schroeder

In the news



Vulnerability in Office 365 allows unauthorised Administrator access

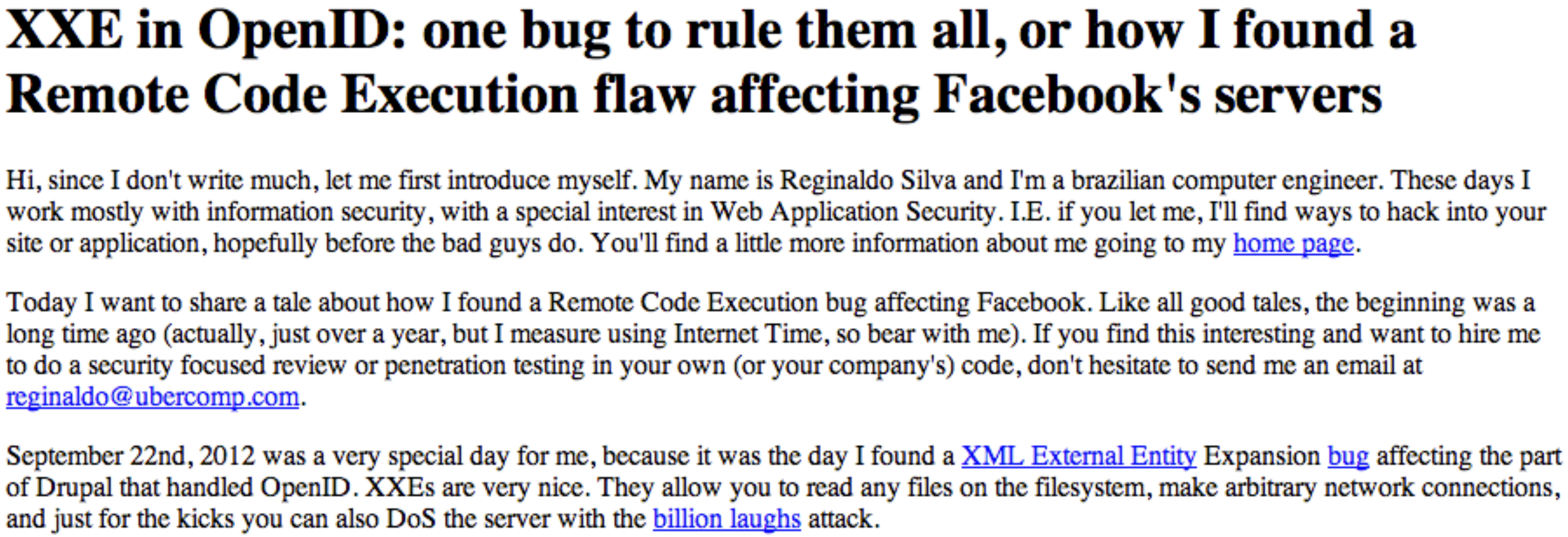
Posted on January 15, 2014 by Alan Byrne

I recently discovered a serious Cross Site Scripting (XSS) vulnerability in Microsoft Office 365 whilst doing a security audit of our own Microsoft Office 365 Reporting Application.

Any person with a mailbox in a company using Office 365 could exploit this vulnerability to obtain full Administrative permissions over their entire company's Office 365 environment using just a few lines of JavaScript.

https://www.cogmotive.com/blog/office-365-tips/vulnerability-in-office-365-allows-unauthorised-administrator-access

In the news



XXE in OpenID: one bug to rule them all, or how I found a Remote Code Execution flaw affecting Facebook's servers

Hi, since I don't write much, let me first introduce myself. My name is Reginaldo Silva and I'm a brazilian computer engineer. These days I work mostly with information security, with a special interest in Web Application Security. I.E. if you let me, I'll find ways to hack into your site or application, hopefully before the bad guys do. You'll find a little more information about me going to my home page.

Today I want to share a tale about how I found a Remote Code Execution bug affecting Facebook. Like all good tales, the beginning was a long time ago (actually, just over a year, but I measure using Internet Time, so bear with me). If you find this interesting and want to hire me to do a security focused review or penetration testing in your own (or your company's) code, don't hesitate to send me an email at reginaldo@ubercomp.com.

September 22nd, 2012 was a very special day for me, because it was the day I found a XML External Entity Expansion bug affecting the part of Drupal that handled OpenID. XXEs are very nice. They allow you to read any files on the filesystem, make arbitrary network connections, and just for the kicks you can also DoS the server with the billion laughs attack.

http://www.ubercomp.com/posts/2014-01-16_facebook_remote_code_execution

Injection vulnerabilities

General class of vulnerabilities

- Application expects input from user to be used in some “command”
- Input is not properly validated
- Command semantics changes after plugging in user's input

Several instances:

- SQL injection
- Shell command injection
- XPATH injection

SQL INJECTION

SQL injection basics

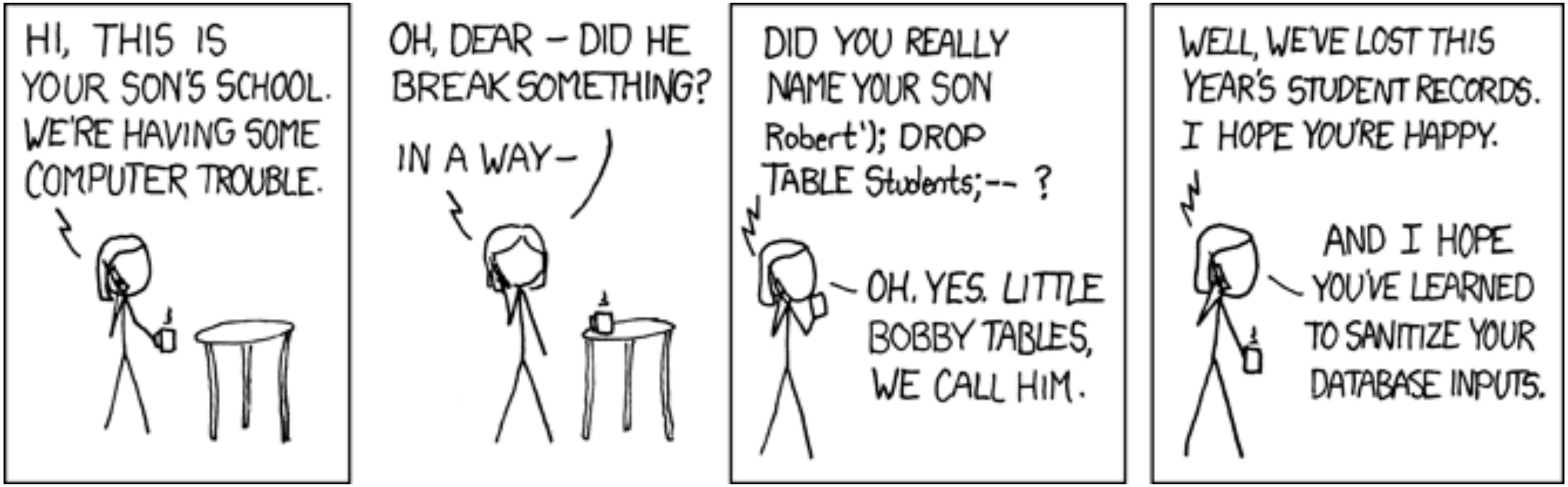
- SQL queries are built using (unsanitized) user's data

```
String q = "SELECT user, pwd FROM users “
   + "WHERE user = '"
   + request.getParameter("user") + "' ";
stmt.executeQuery(q);
```

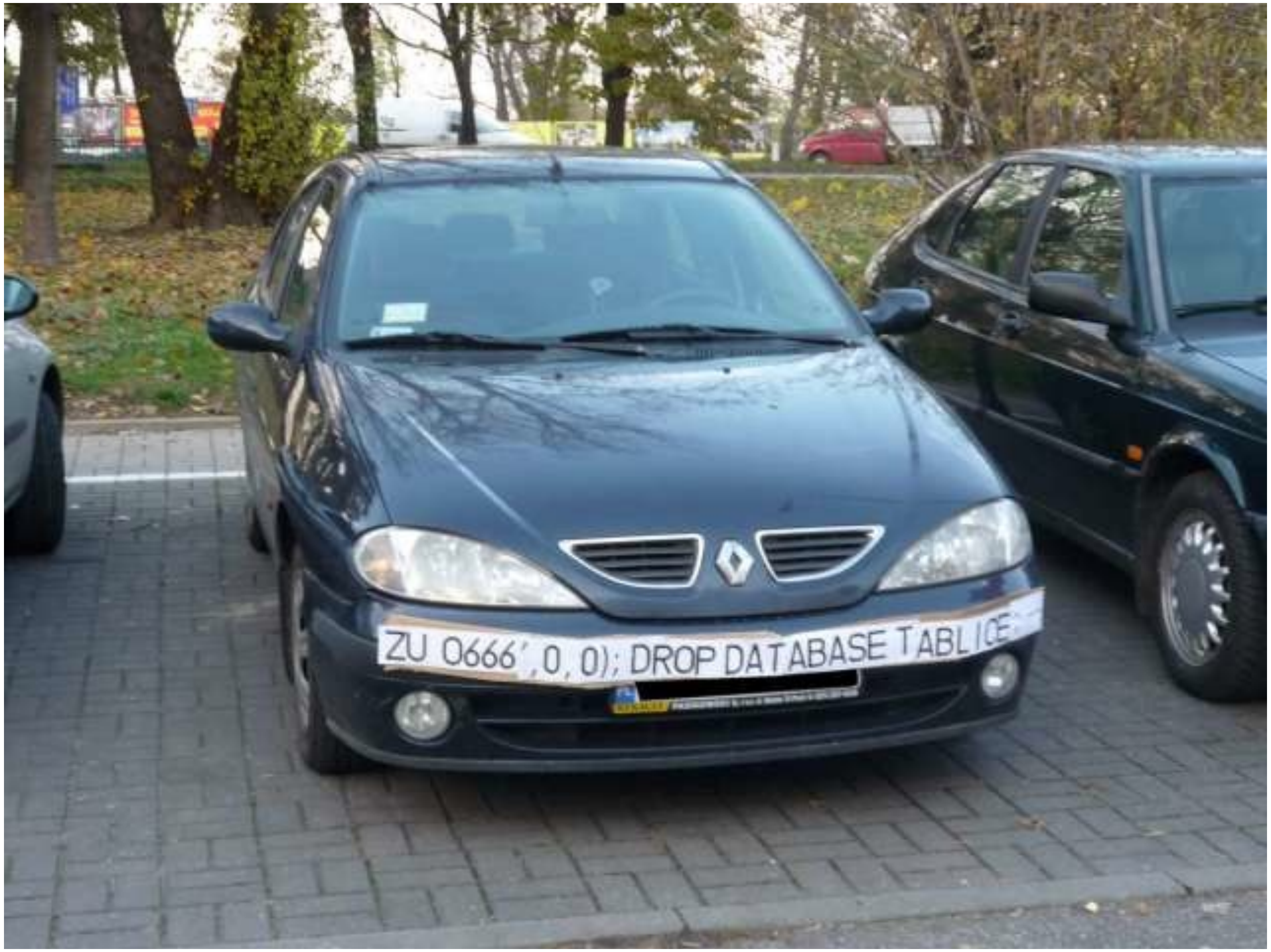
If the attacker provides as parameter special characters such as ‘ (tick), -- (comment), + (space), % (wildcard), it is possible to:

- Modify queries in an unexpected way
- Probe the database
- Run commands (e.g., using xp_commandshell in MS SQL Server)

SQL injection folklore


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 injection folklore



Defenses

- The application is not vulnerable if it uses prepared statements

```
import java.sql.PreparedStatement;

PreparedStatement authQuery =
  conn.prepareStatement(
    "SELECT user, pwd FROM users WHERE user = ?");
authQuery.setString(1,
                    request.getParameter("user"));
authQuery.executeQuery();
```

Finding SQL injection (whitebox)

1. Identify all queries that do not use prepared statement and accept input from user
2. Trace back from the queries to identify sanitizations applied to user's input (if any)
3. Determine if sanitization is effective

What strategy is this?

Finding SQL injection (blackbox)

- Provide the application specially-crafted values and check if they cause errors
  - ’, ”, #
- Inject expression (typically a tautology) and check if it is interpreted:
  - `user=' OR 1=1 #`

Exploitation technique: conditional responses

We leverage the SQL injection to ask boolean questions to the server, e.g.,

- Are we running as root?
- Is the first letter of the current database 'a’?

Steps:

- Establish baseline: determine what response is provided by the application for a true question (“true page”) and for a false question (“false page”) inject question
- Compare result with baseline:
  did we obtain a true page or a false page?

Exploitation technique: conditional responses

Scenario:

- Assume there is a SQL injection on product_id parameter: `/view?product_id=N`
- True page: “Details about product...”
- False page: “No information about the product you searched”

Injections:

- `product_id=42 AND user() = "root"`
- `product_id=42 AND substring(database(), 1, 1) = 'a'`

Exploitation technique: time-based

- Are we running as root?

```
SELECT IF (USER()="root", SLEEP(5),
1);
```

- Is the first letter of the user 'a’?

```
SELECT IF(SUBSTRING(USER(), 1, 1) =
'a', SLEEP(10), 2);
1 row in set (0.00 sec)
```

- Is the first letter of the user 'n’?

```
SELECT IF(SUBSTRING(USER(), 1, 1) =
'n', SLEEP(10), 2);
1 row in set (10.00 sec)
```

Exploitation technique: bypassing filters

Filter function is used by the programmer to “sanitize” user input:

- Accept only characters from a vetted set (whitelisting)
- Remove dangerous characters/keywords (blacklisting)

Sometime it is possible to bypass such filters...

(Which one is better according to our security principles?)

Filter: removes tick (‘)

Goal:

```
substring(user(),1,1) = ’r’
```

Bypass:

```
substring(user(), 1, 1) = char(0x72)
```

Filter: removes SQL keywords

Bypass techniques:

- Case sensitivity: SeLecT
- Comments: SEL/**/ECT

Filter: removes whitespace

Goal:

```
SELECT NOW()
```

Bypass:

```
SELECT/**/NOW()
```

Filters: removes tautology

Goal:

`1=1`

Bypass:

`RAND() > 0.0001`

(quasi-tautology)

Filters: lots more

Eduardo Vela and David Lindsay,

Our Favorite XSS Filters/IDS and how to Attack Them

Blackhat 2009

They show how to bypass filters in ModSecurity, PHPIDS, IE8, and NoScript

OTHER INJECTIONS

Command injection

- Application constructs shell command using user- provided input without proper sanitization
- The attacker can execute shell commands on the targeted system with the privileges of the vulnerable application
- Two variants:
  - user input as argument of a single, fixed program: `exec("nslookup $input")`
  - user input passed entirely and directly to shell: `exec($input)`

Vulnerable DNS lookup

```
$hostname = $_POST["hostname"];
$command = '/usr/bin/nslookup ' .
$hostname; system($command);
```

```
google.com; rm -rf
```

A little bit of history

Vulnerability in phf: CVE-1999-0067

Sanitization routine invoked before passing arguments to popen:

```
void escape_shell_cmd(char *cmd) {
  register int x,y,l;
  l=strlen(cmd);
  for(x=0;cmd[x];x++) {
    if(ind("&;`'\"|*?~<>^()[]{}$\\",cmd[x]) != -1){
```

What about \n?

```
phf?Qalias=%0A/bin/cat%20/etc/passwd
```

```
    }
  }
}
```

XPATH injection

- Application constructs XPATH query using user-provided input without proper sanitization
- XPATH query language to programmatically address parts of XML documents
- The attacker can discover the structure of the underlying XML data or access data that he should not have access to
- Unlike SQL, XPATH is standard

  → no need to account for different backends

XPATH injection example

```
<app>
    <conf>
        <db>
            <password>secret</password>
        </db>
    </conf>
    <route prefix="/users">
        <target>/var/www/app/users/list.jsp</target>
    </route>
</app>
```

Query: `"//route/[@prefix='" + request.getRequestURI() + "']"`

```
/users/']/../conf/db/password['foo
```

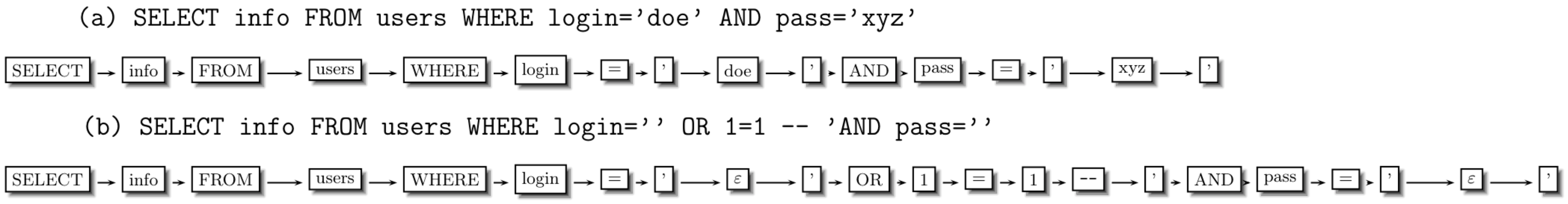
(Test it here)

Avoidance, detection, and prevention

DEFENSES

Parametrization

- Essence of injection attacks: user-provided input changes the structure and semantics of the query



(a) `SELECT info FROM users WHERE login=’doe’ AND pass=’xyz’`

(b) `SELECT info FROM users WHERE login=’’ OR 1=1 -- ’AND pass=’’`

Parametrization

Separate variable parameters (“out-of-band”)

- Use placeholders for variable parameters
- Bind parameter to a specific data type before issuing query/command

Parametrized commands

```
import MySQLdb

conn = MySQLdb.connect()
c = conn.cursor()
c.execute("""SELECT *
          FROM users
          WHERE user = %s AND
                pass = %s""",
          (input_user, input_pass))
```

Sanitization

- Never trust user input
- Ensure data is valid and good
  - Valid: can be correctly processed (e.g., if you expect XML document as input, check that it is well-formed)
  - Good: the expected kind of data (XML document validates a required schema)
- Use whitelisting whenever possible
- Use standard, built-in functions to perform sanitization (`strip_tags`, `htmlentities`, etc.)

Static analysis

- Set of techniques to automatically analyze a program without executing it
- Useful to answer questions such as:
  - which values can variable x have at line 23?
  - is there a command injection in the program?
  - does the program always produce valid HTML?
- Several different techniques:
  - program analysis
  - model checking
  - …

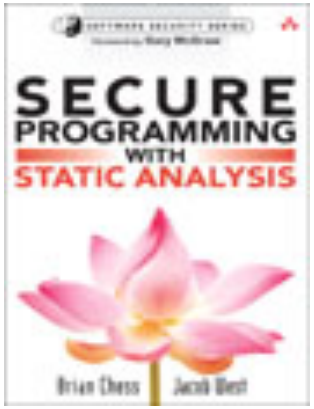
Static analysis

- Limitations:
  - Rice's theorem
  - Memory and time constraints
  - …
- False positives and false negatives

- To know more, read:

  B. Chess and J. West, Secure Programming with Static Analysis

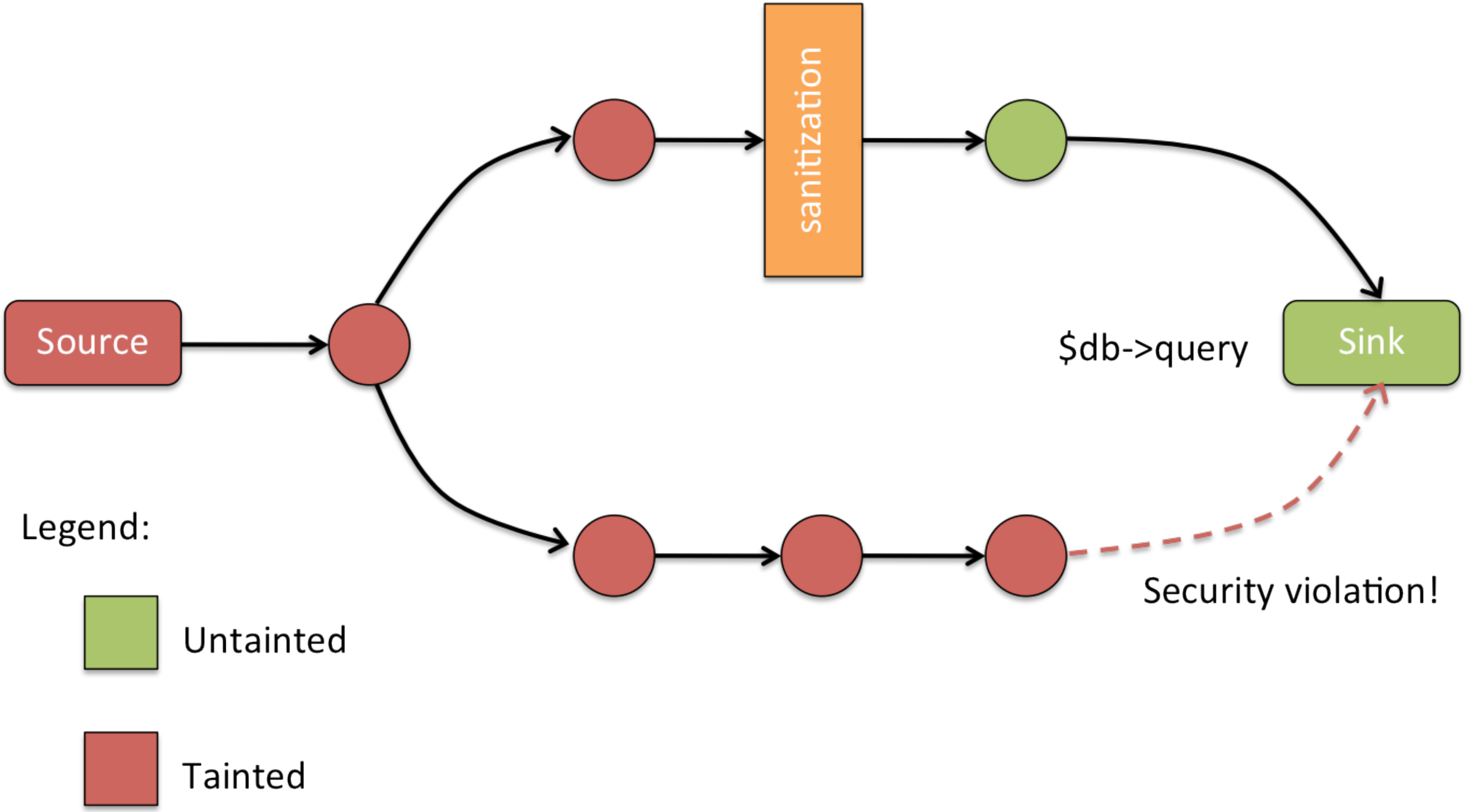
  Pearson, 2007



Taint analysis

- Data objects partitioned into “tainted” and “untainted”
- Detection policy: “tainted objects should not reach certain program locations”
- Ingredients:
  - Sources are program locations that generate tainted data (e.g., request parameters)
  - Sinks are program locations that should not use tainted data (e.g., SQL queries)
  - Propagation rules model how program statements affect objects taintedness (e.g., copying source input to variables)

Taint analysis



Source

sanitization

$db->query

Sink

Security violation!

Legend:

🟩 Untainted

🟥 Tainted

Take away points

- Several instances of injection attacks
- Techniques to avoid, detect and prevent them

Next time

- Buffer overflows