Class 09

CSC 600 Advanced Seminar
Intro to Web Security

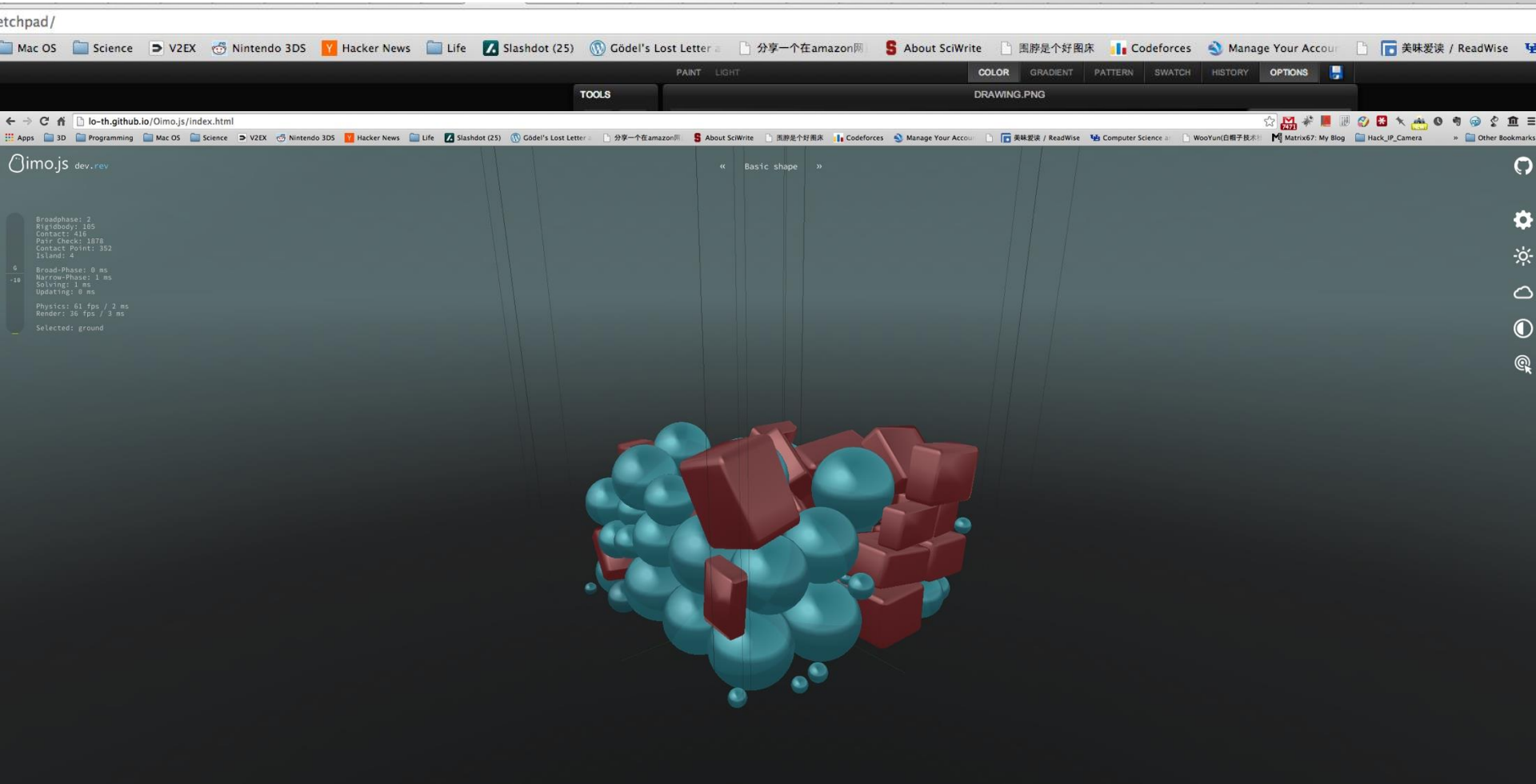
Dr. Si Chen (schen@wcupa.edu)

The Evolution of Web Applications

- In the early days of the Internet, the World Wide Web (WWW) consisted only of websites.
  - essentially information repositories containing static documents.

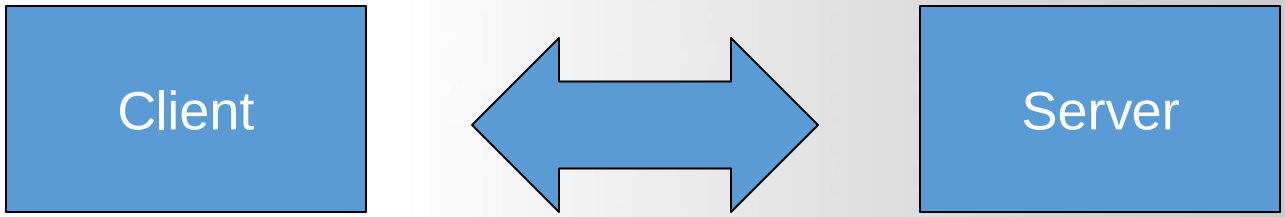
The Evolution of Web Applications

- Today, the majority of sites on the web are in fact applications.
  - Highly functional
  - Rely on two-way flow of information between the server and browser.



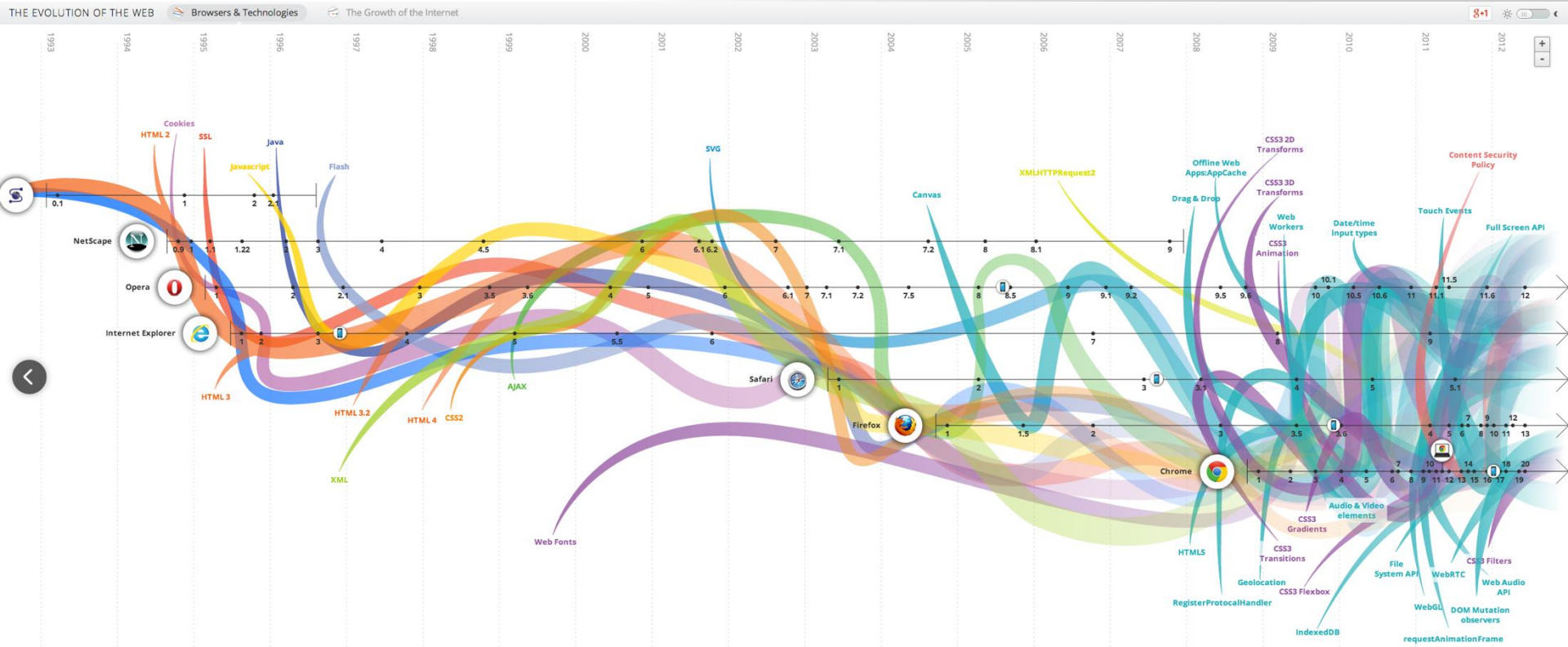
THE WEB IS SIMPLE

- Hyper Text Transfer Protocol (HTTP)
- Designed to allow remote document retrieval
- Simple client server model:



The Reality

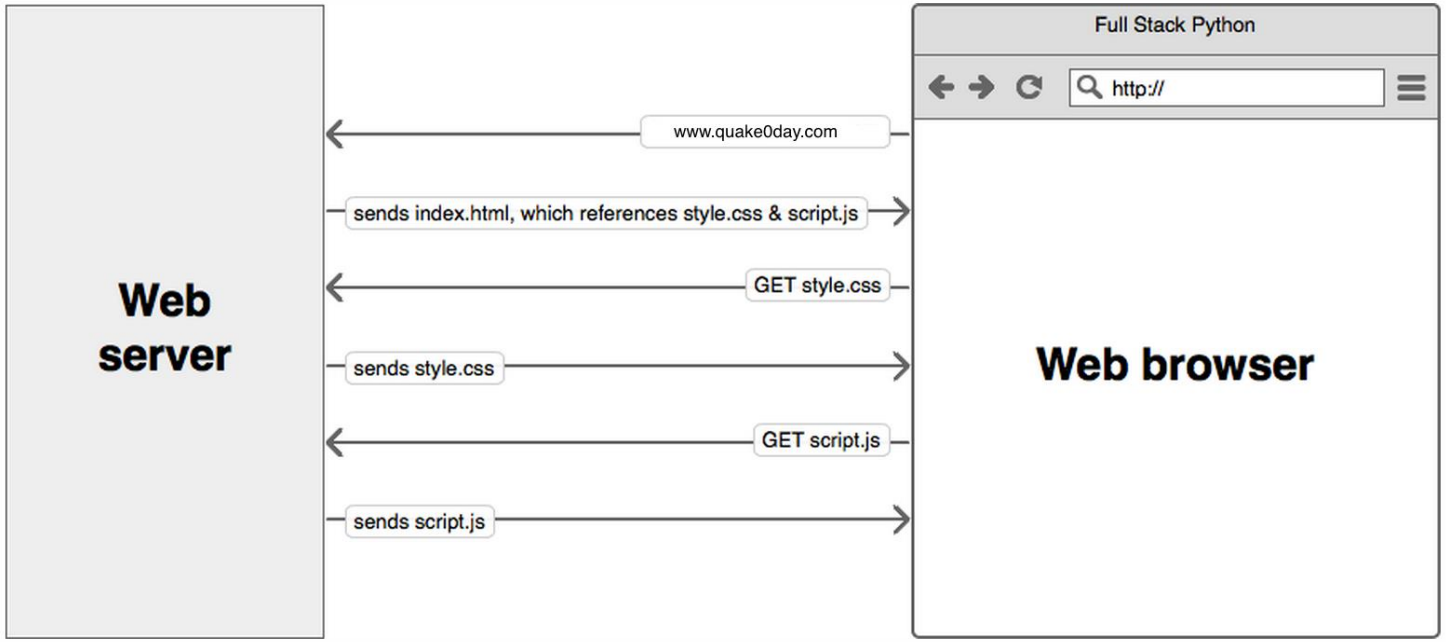
- Web application security is massively complex.
- Constant evolving field
  - ES6, ES8, HTML5, CSS3, AJAX…



http://www.evolutionoftheweb.com/

Typical Web Application Stack

- Browser (client)
- HTTP over TCP/IP
- Server
  - Operating system
  - Web Server
  - Scripting Language
  - Database or persistence layer



Just the client

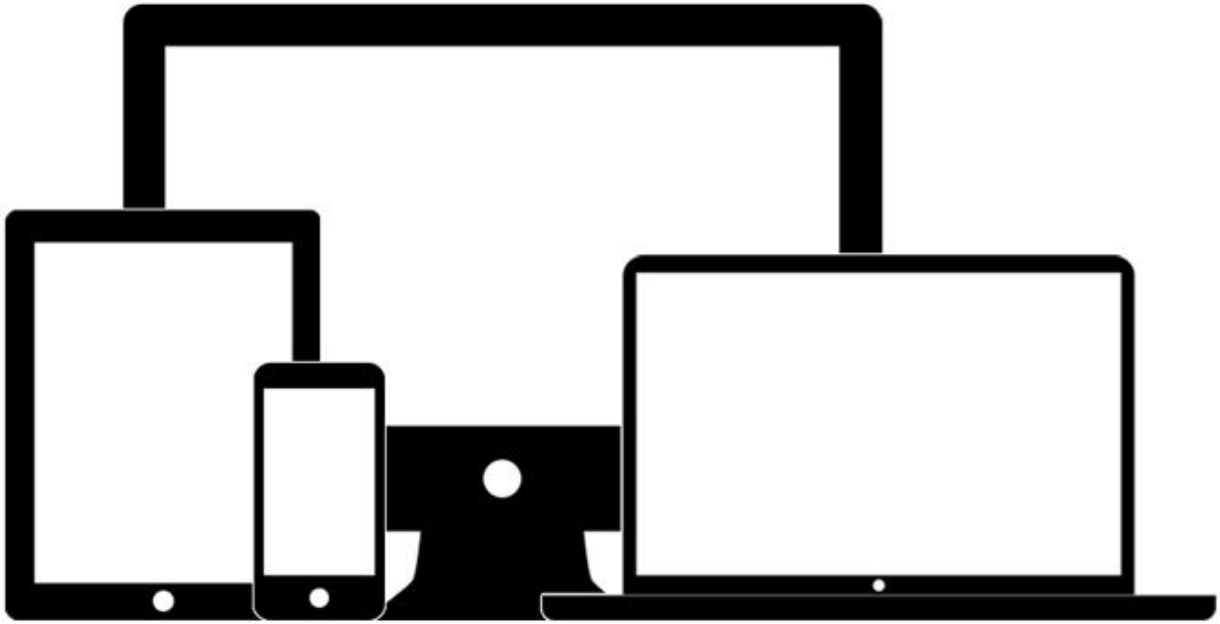
- Many different clients, all implementing differently (Chrome, Firefox, Edge, IE, Safari, Opera, etc…)
- The breakdown of the client-server divide
  - The functional boundaries between client and server responsibilities were quickly eroded

JavaScript

1. JavaScript allows for client side programming (responsive user interface (UI))
2. Plug-in's allow for store data locally (jStorage)
3. AJAX allows display multiple HTML sources in one page

Fertile Ground

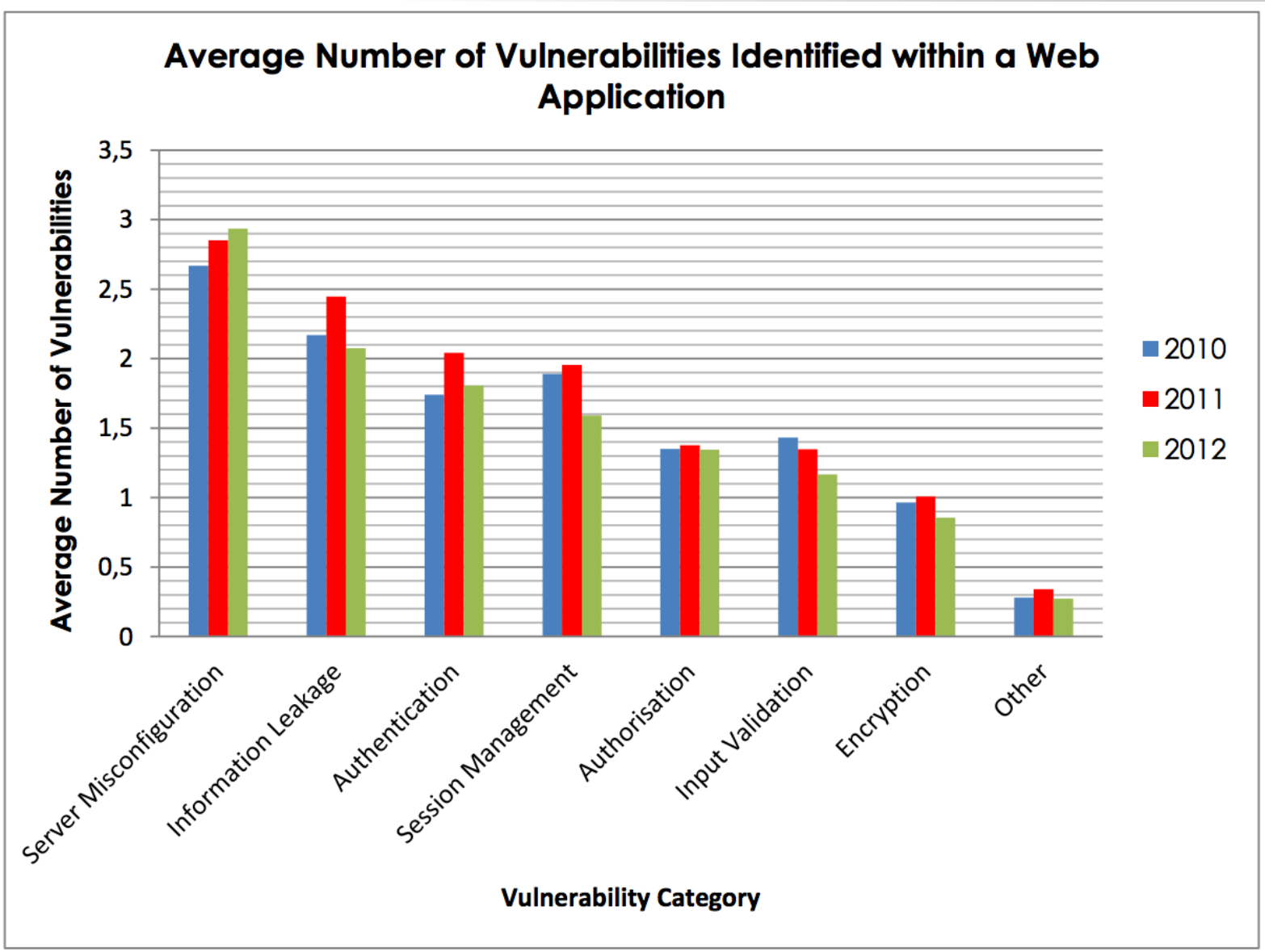
- Web application security is massively complex in reality:
  - Security researchers specialize in specific portions of the stack
  - Protocols and specs exist but aren't implemented uniformly
  - The platforms are changing
    - Smartphone, tablets, embedded systems, etc…



What's Worse

- In the browser world, the separation between high-level data objects (documents), user-level code (applications) is virtually nonexistent.
- Firewalls become irrelevant as everything flows over port 80 (http), 443 (https)
- Web is becoming the default content and application deliver mechanism

Average Number of Vulnerabilities Identified within a Web Application



http://www.contextis.com/files/Web_Application_Vulnerability_Statistics_-_June_2013.pdf

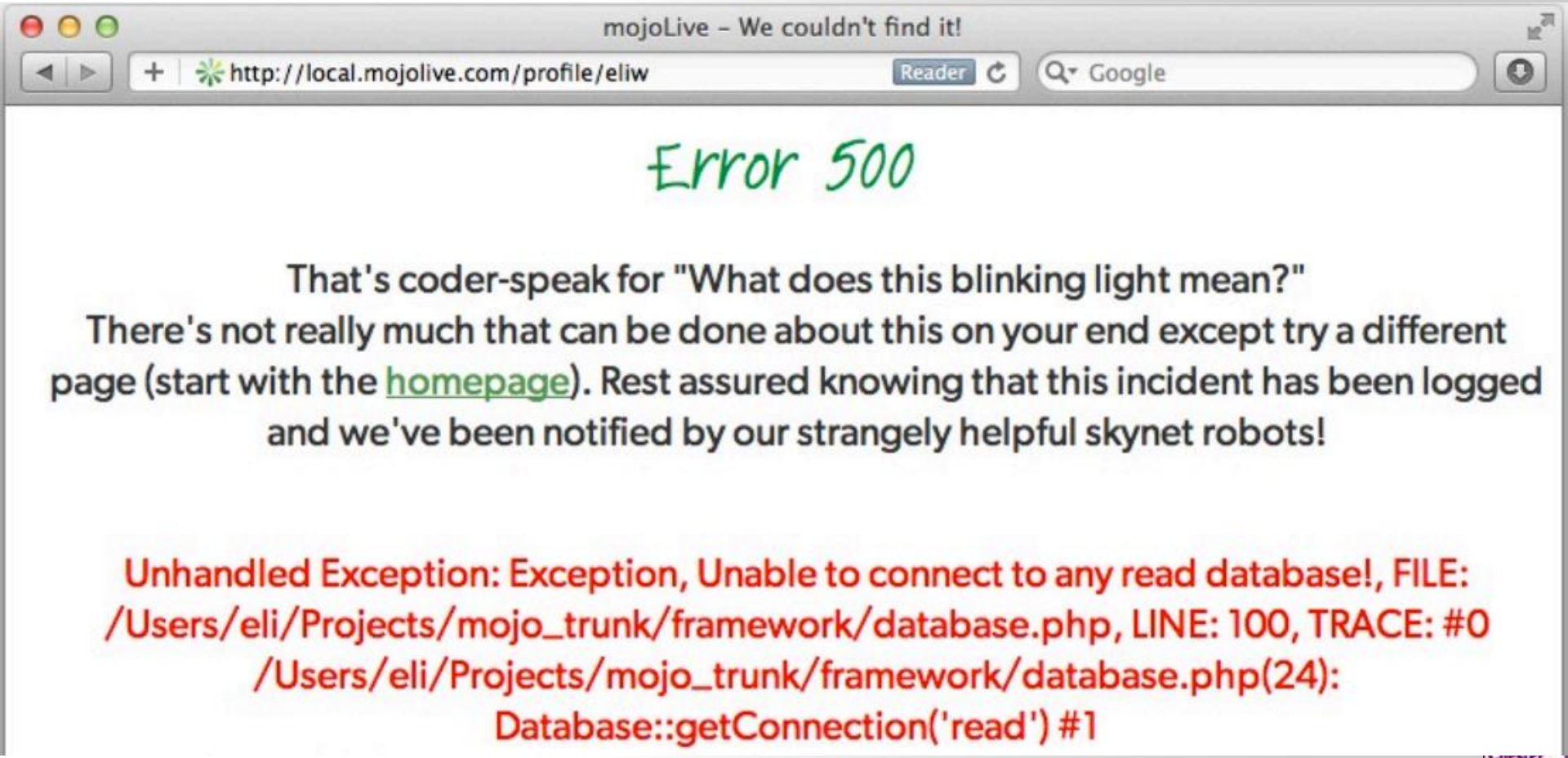
Categories of Vulnerability

Category	Description
Server configuration	Insecure server configuration settings that result in security vulnerabilities
Information leakage	Information leaked by the application that could be used by an attacker to help mount an attack
Authentication weaknesses	Issues related to the application's authentication mechanism that could be exploited by an unauthenticated attacker to gain or assist in the gaining of authenticated access
Session management weaknesses	Session management issues that could allow an attacker to hijack or assist in the hijacking of other users' sessions
Authorisation weaknesses	Issues concerning access controls that could allow an attacker to perform either horizontal or vertical privilege escalation
Input validation weaknesses	Issues created by weaknesses in input validation processes.
Encryption vulnerabilities	Issues that concern the confidentiality of data during transport and in storage
Other	Any other issues identified that do not fit into the categories listed above

http://www.contextis.com/files/Web_Application_Vulnerability_Statistics_-_June_2013.pdf

Information Leakage

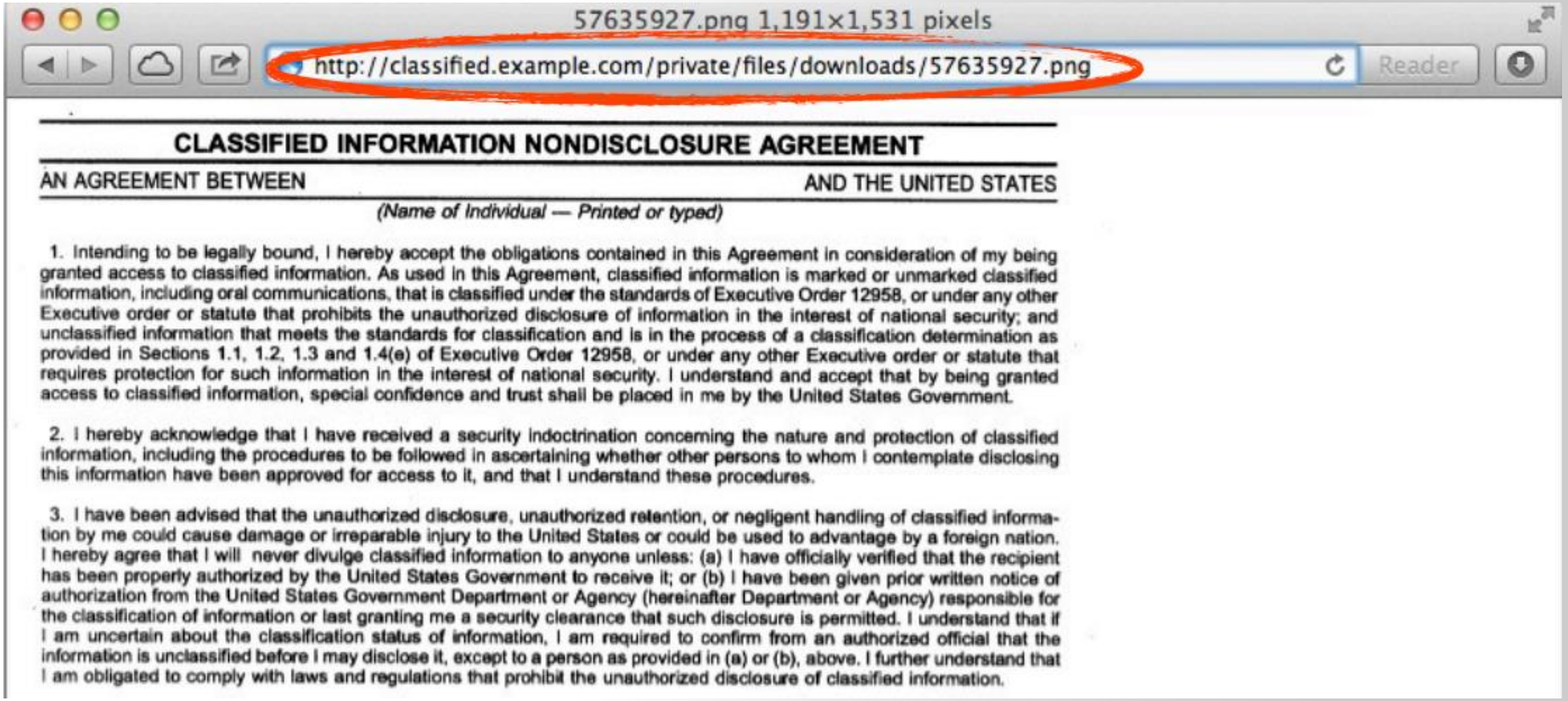
Visible Error Handling



http://Local.mojolive.com/profile/eliw

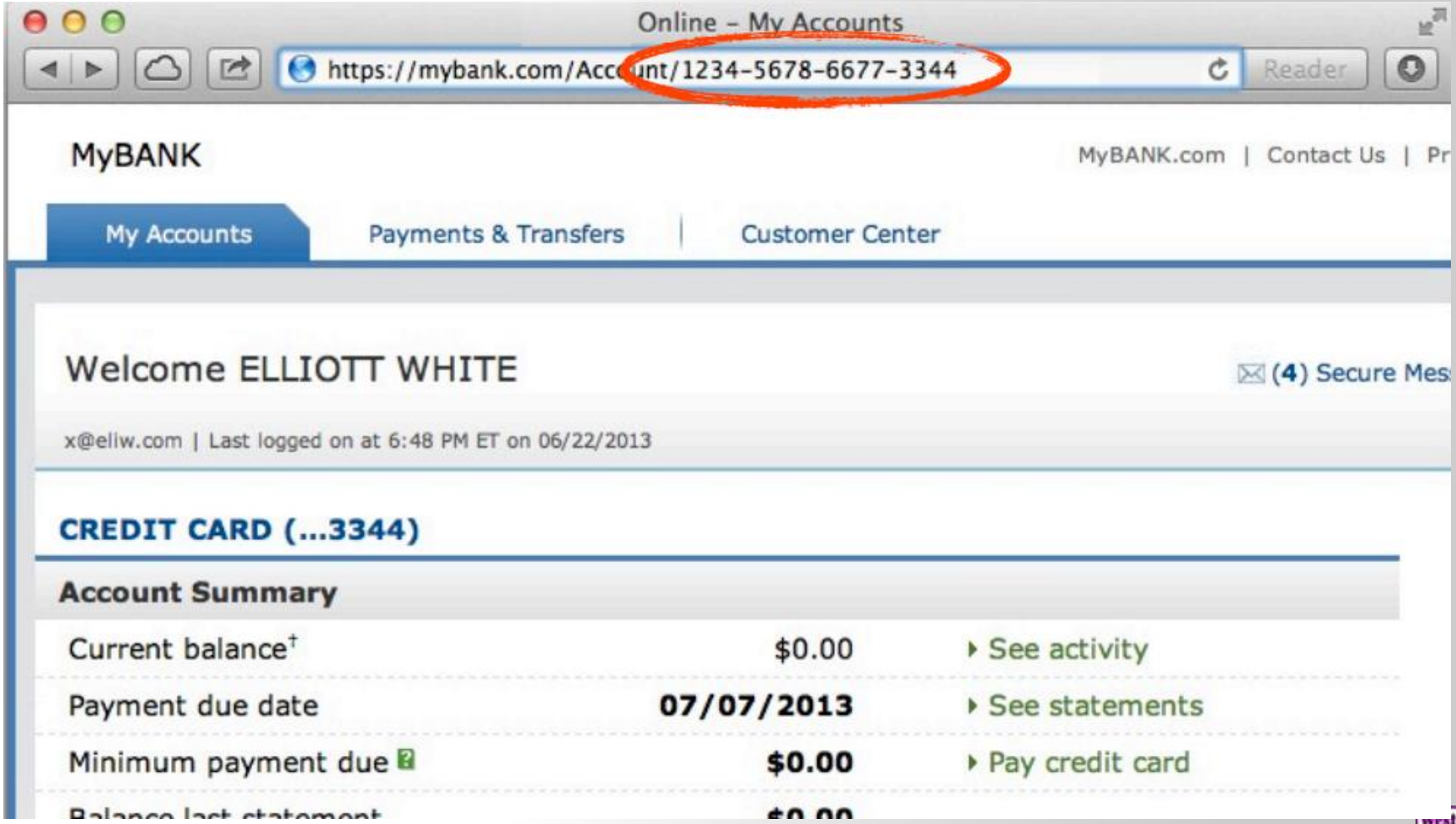
Authentication Weakness

Direct URL access to a protected file



Authentication Weakness

Ability to URL-hack to access unauthorized data.

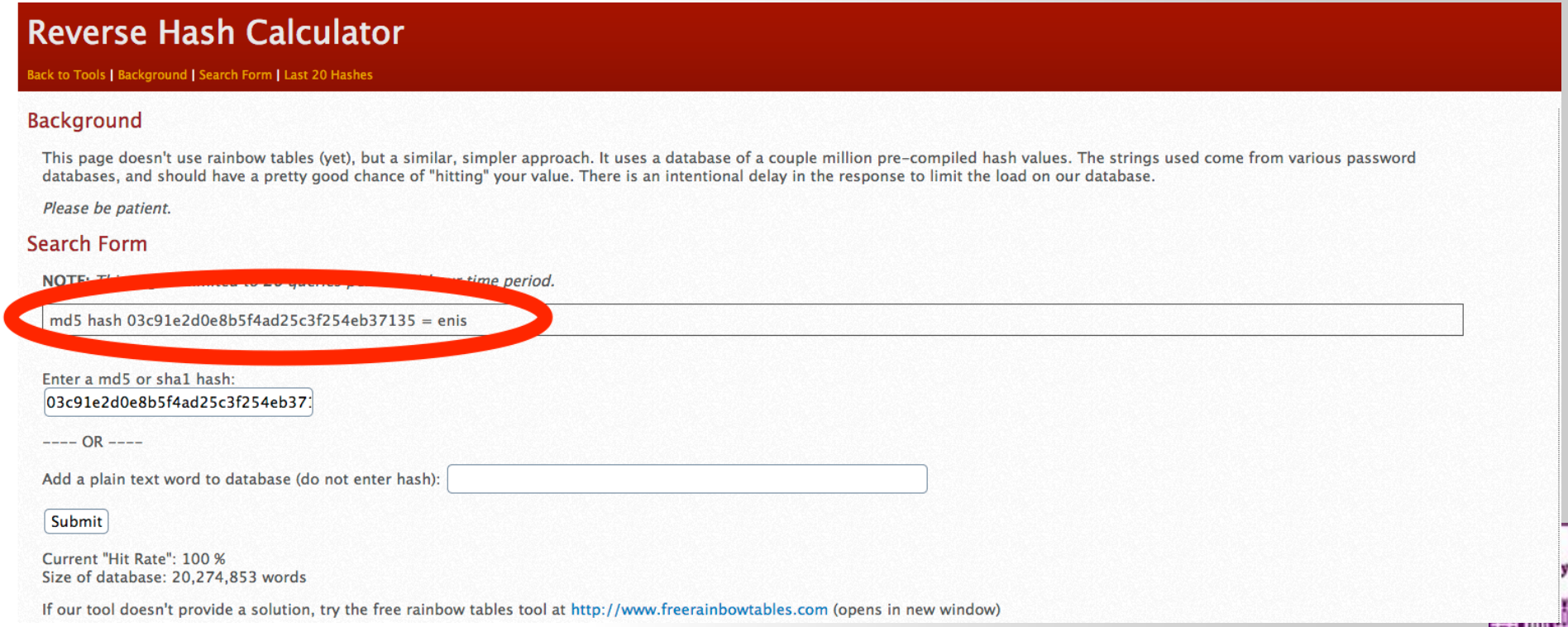


Eli White, *Web Security and* You,page7

Encryption Vulnerabilities

Low Security Hashes

1. Don’t just use MD5 (use SHA256, Blowfish,etc…) Even SHA-1 is better
2. Always salt your hashes

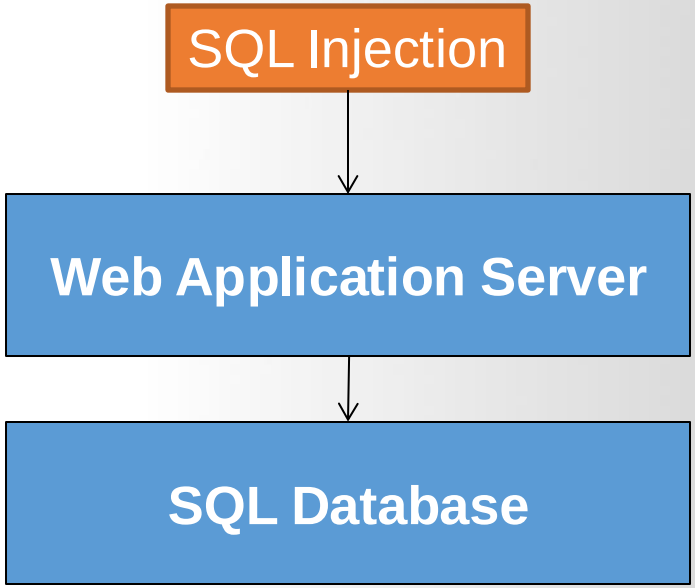


Various Attack Vectors

Now moving on to true ‘attacks’….

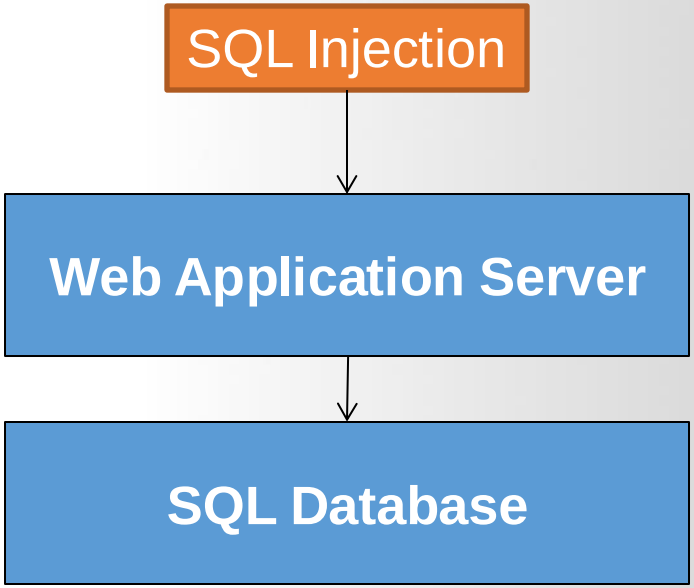
Attacking Data Stores: SQL Injection

- **SQL** stands for Structured Query Language.
- **SQL** is used to communicate with a database.



Attacking Data Stores: SQL Injection

- It is common to build an SQL Database query based in part on a user submission.
  - User submits a login request, we need to check the database for a matching account.
- Malicious user know we will be building an SQL query.
  - They can attempt to confuse the Database server by putting in special characters



Attacking Data Stores: SQL Injection

- A user having the ability to send data that is directly interpreted by your SQL engine.

The Security Hole:

```
<?php
 $pdo->query("SELECT * FROM users
  WHERE name = '{$_POST['name']}' AND pass = '{$_POST['pass']}'");
?>
```

The Attack:

```
<?php
 $_GET['name'] = "' or 1=1; //";
?>
```

```
<?php
 $pdo->query("SELECT * FROM users
  WHERE name = ' or 1=1; //AND pass = '{$_POST['pass']}'");
?>
```

Always "True"

Attacking Data Stores: SQL Injection

- Sanitize every value received from the user.
  - Make sure there is no funny business going on
  - Makes any string safe to put in a query

The Solution:

```php
<?php
 $query = $pdo->prepare("SELECT * FROM users WHERE name = ? AND pass = ?");
 $query->execute(array($_POST['name'], $_POST['pass']));
?>
```

OR:

```php
<?php
 $name = $pdo->quote($_POST['name']);
 $pass = $pdo->quote($_POST['pass']);
 $pdo->query("SELECT * FROM users WHERE name = {$name} AND pass = {$pass}");
?>
```

Attacking Data Stores: SQL Injection

Logon page that shows an SQL injection string

http://sqlmag.com/content/content/43012/figure_01.gif

Other Injections

- **Command Injection:** The user being able to inject code into a command line
- **Unchecked File Uploads:** The user being allowed to uploadan executable file.
- **Code Injection:** User being able to directly inject code.

```php
<?php
 $retval = exec('echo “$line” >> logfile.txt');
?>
```

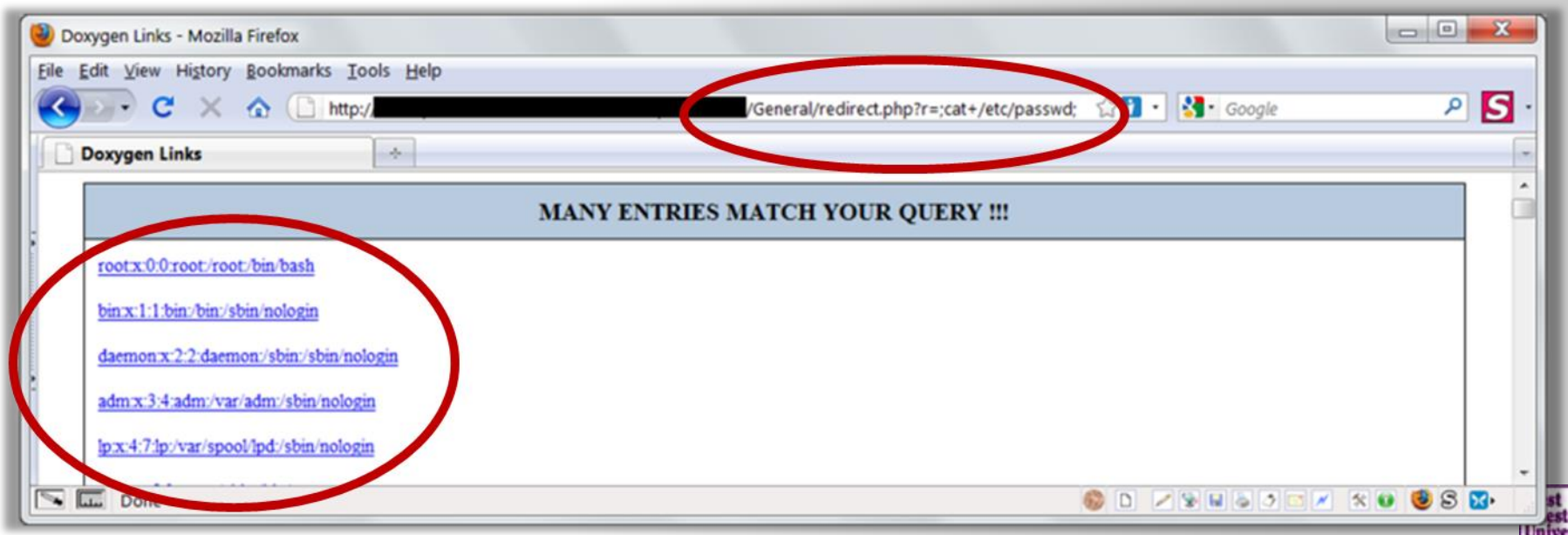
becomes

```php
<?php
 $retval = exec('echo ""; rm -rf *; echo "" >> logfile.txt');
?>
```


https://security.web.cern.ch/security/recommendations/images/command_injection.png

Other Injection

- **Command Injection:** The user being able to inject code into a command line
- **Unchecked File Uploads:** The user being allowed to uploadan executable file.
- **Code Injection:** User being able to directly inject code.



https://security.web.cern.ch/security/recommendations/images/command_injection.png

Real World Attacks

- https://www.youtube.com/watch?v=Qb8-0zGiE7A

Real World Attacks

```
[exploit@zer0-day code]$ ./bxcp http://www.bxcp.com gh0st:66
h0n1g:b5aa2b48dbea8988e09addd46b4cbf68
dasB&ouml;se:e20adc3949ba59abbe56e057f20f8851
hajo:811fe6c28526e72589981c923d51821e
Dinniz:a36fe0898c0a91541d7bbcd24f853091
kevinek:345d52aee878b0bab3c084ef179474ef
Xale:318d675f09cb6b9d7944088990eb6213
PrivateMike:23c14f311a60486b36f79f3bc962be36
Species8472:336327a91ffaa7d53f1f0b42424c0c3g
exxistenz:313edbdddc4450ffc7bde632c122d961
DxBlueIce:c1e26c579de0f76d181168a7dc5f9711
[exploit@zer0-day code]$ ./bxcp http://www.bxcp.com gh0st:66 > dump.pwd
```

MD5 Result

Md5: b5aa2b48dbea8988e09addd46b4cbf68

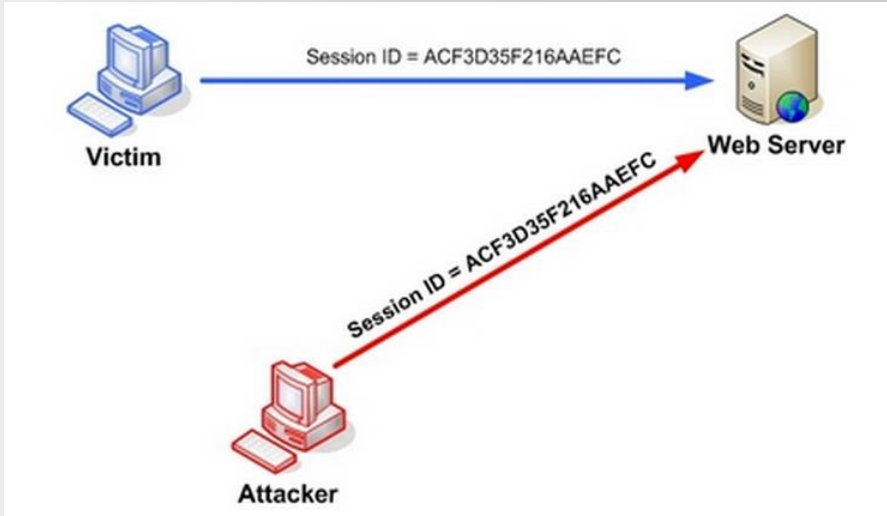
Result: voodoo

Attacking Session Management: Session Hijacking

- The HTTP protocol is essentially stateless. It is based on a simple request-response model.
- Majority of web applications allow you to register and log in. To implement this functionality, web apps need to use the concept of a ***session***.
- The vulnerabilities that exists in session management mechanisms largely fall into two categories:
  - Weakness in the generation of session tokens
  - Weakness in the handling of session tokens throughout their life cycle

Attacking Session Management: Session Hijacking

- One user ‘becoming’ another by taking over their session via impersonation.
  - Avoid “session Fixation”, don’t use URL cookies for your sessions
  - Always regenerates Session IDs on a change of access level
  - Save an anti-hijack token to another cookie & session. Require it to be present & match. Salt on unique data (such as User Agent)

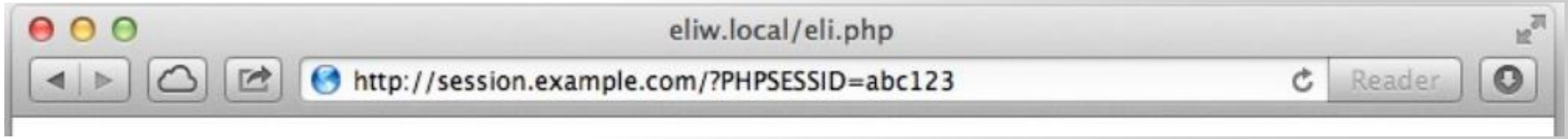


https://www.owasp.org/index.php/Session_hijacking_attack

Attacking Session Management: Session Hijacking

A user being able to provide a known session ID to another user.

The Attack:



The Solution:

Don’t use cookies for your sessions.

Protect from more complicated fixation attacks, by regenerating sessions on change of access level.

Use anti-hijack measures to ensure user is legit

Attacking Users: Cross-Site Scripting (XSS)

- The attacks we have considered so far involve directly targeting the sever-side application.
- Many of these attacks do impinge upon other users, such as SQL injection. But the attacker's essential methodology was to **interact with the server** in unexpected ways to perform unauthorized actions and access unauthorized data.
- Cross-site scripting, however, are in a different category.
- The attacker's primary target: **the application's other users.**

Basic idea: A user sending data that is executed as script

Attacking Users: Cross-Site Scripting (XSS)

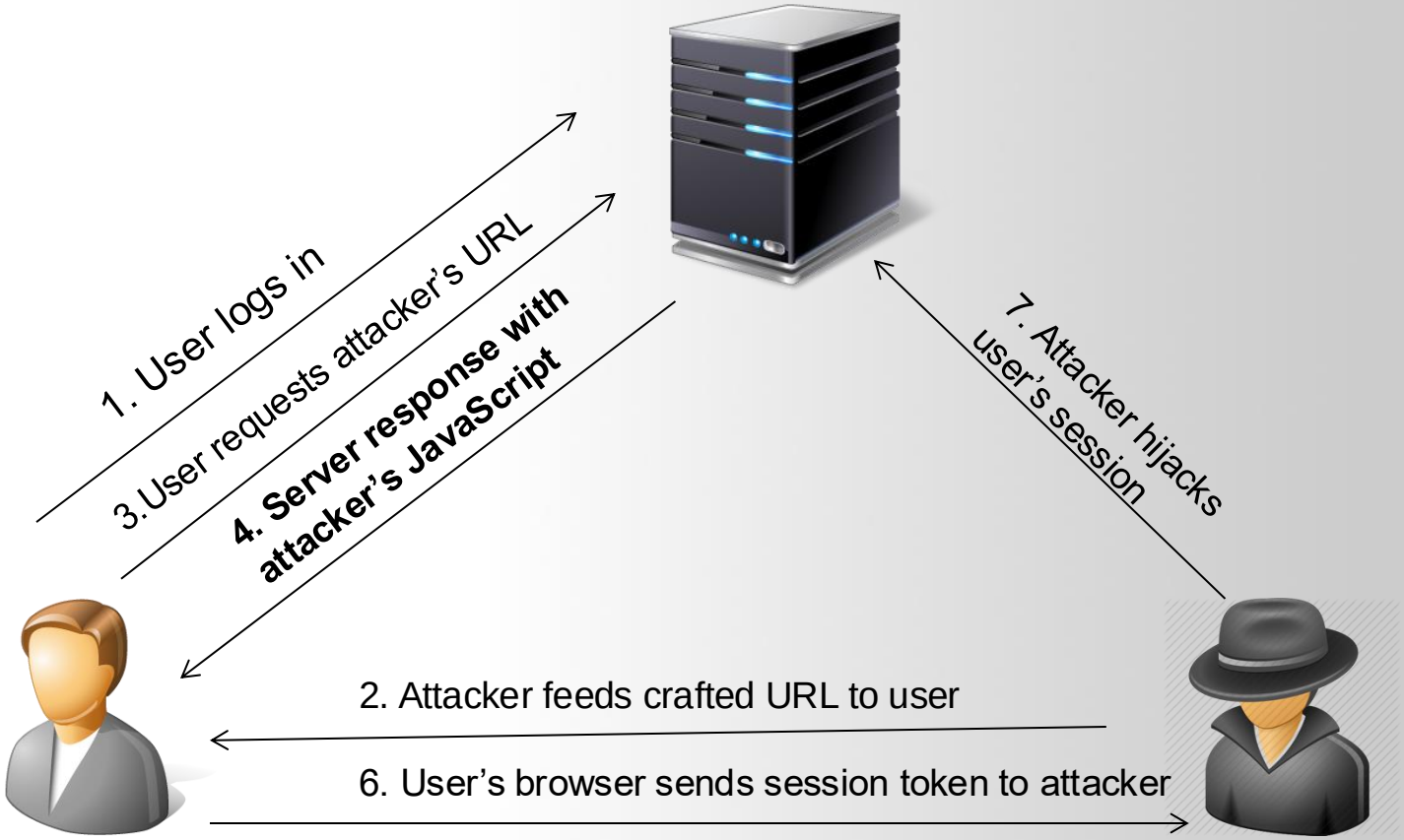
- XSS vulnerabilities come in various forms and may be divided into three varieties: reflected, stored, and DOM-based.
- They have important differences in how they can be identified and exploited.
- In all cases: **Everything** from a user is suspect (forms, user-agent, headers, etc) when fixing, escape to the situation (HTML, JS, XML, etc) **FIEO** (Filter Input, Escape Output)

- We will examine each variety of XSS in turn.

Attacking Users: Cross-Site Scripting (XSS)

XSS- Reflected XSS:

Directly echoing back content from the user.



Attacking Users: Cross-Site Scripting (XSS)

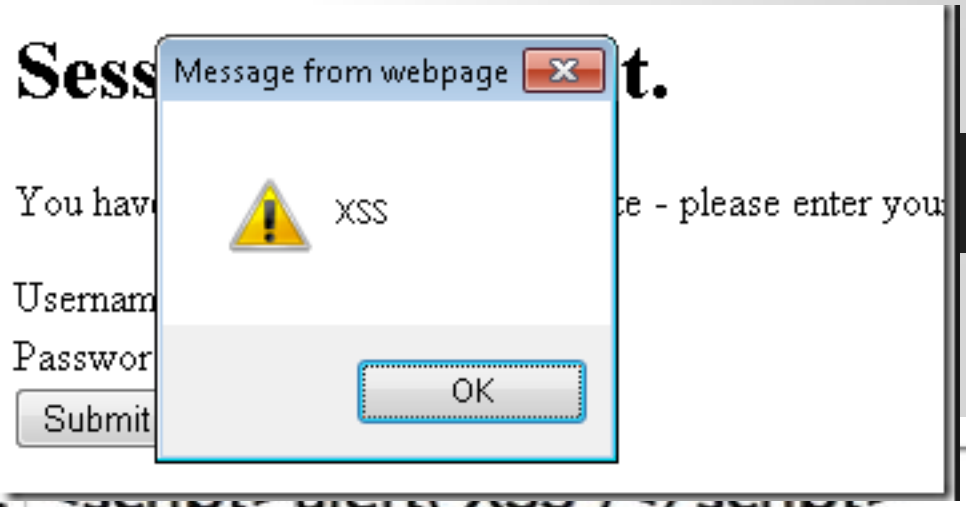
XSS- Reflected XSS:

Directly echoing back content from the user.

The Security Hole

```
<p>Thank you fo                                    rst_name'] ?></p>
```

The Attack:



First Name: Submit

This type of simple XSS bug accounts for approximately 75% of the XSS vulnerabilities that exists in real-world web apps.
It is called reflected XSS because exploiting the vulnerability involves crafting a request containing embedded JavaScript that is reflected to **any user who makes the requests**

XSS- Reflected XSS:

Directly echoing back content from the user.

http://twitter.com/index.php?%75%73%65%72%3D%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%31%32%33%29%3C%2F%73%63%72%69%70%74%3E

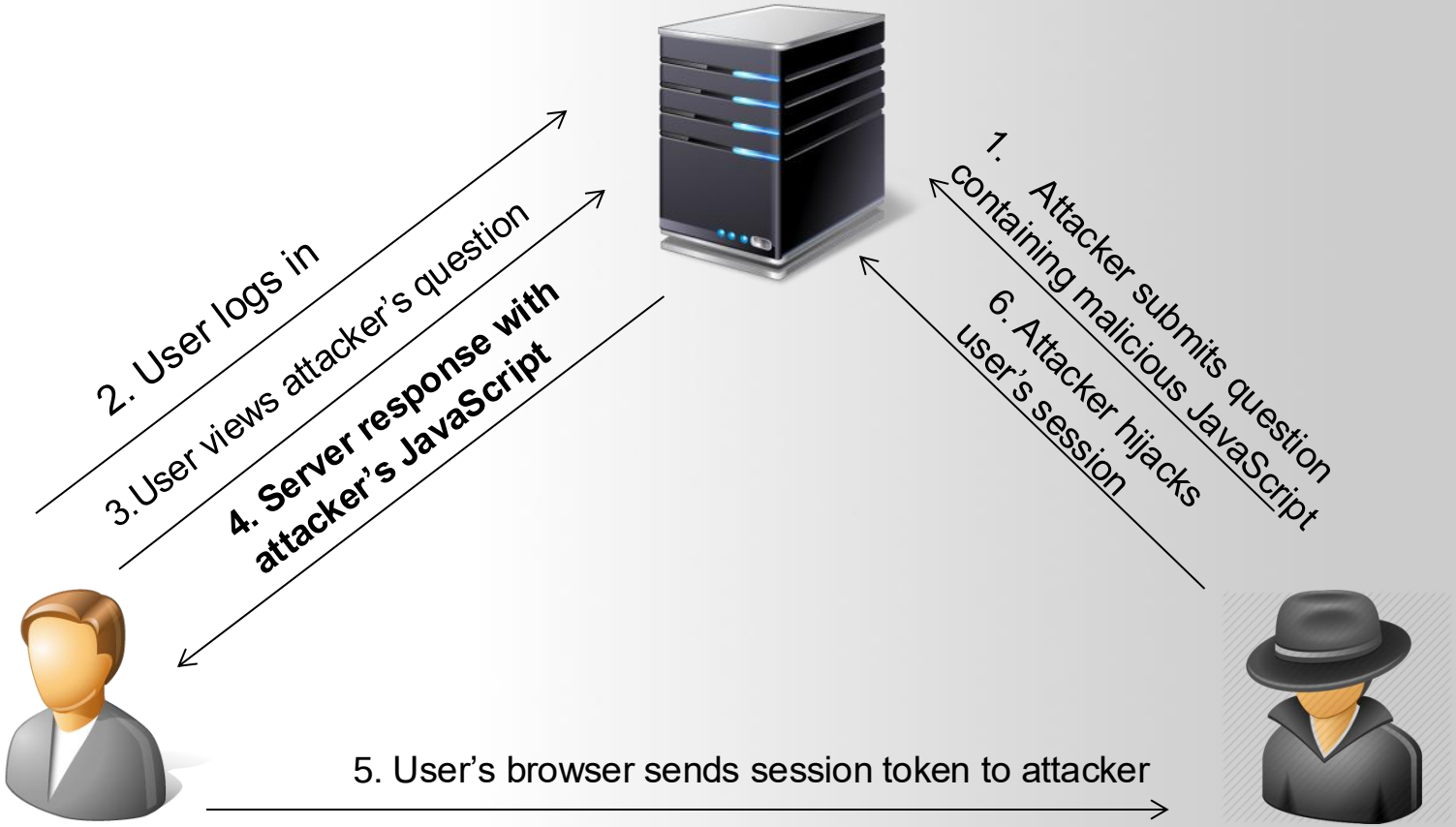
Twitter?

http://twitter.com/index.php?user=<script>alert(123)</script>

Attacking Users: Cross-Site Scripting (XSS)

XSS- Stored XSS:

You store the data, then later display it.



Attacking Users: Cross-Site Scripting (XSS)

XSS- Stored XSS:

You store the data, then later display it.

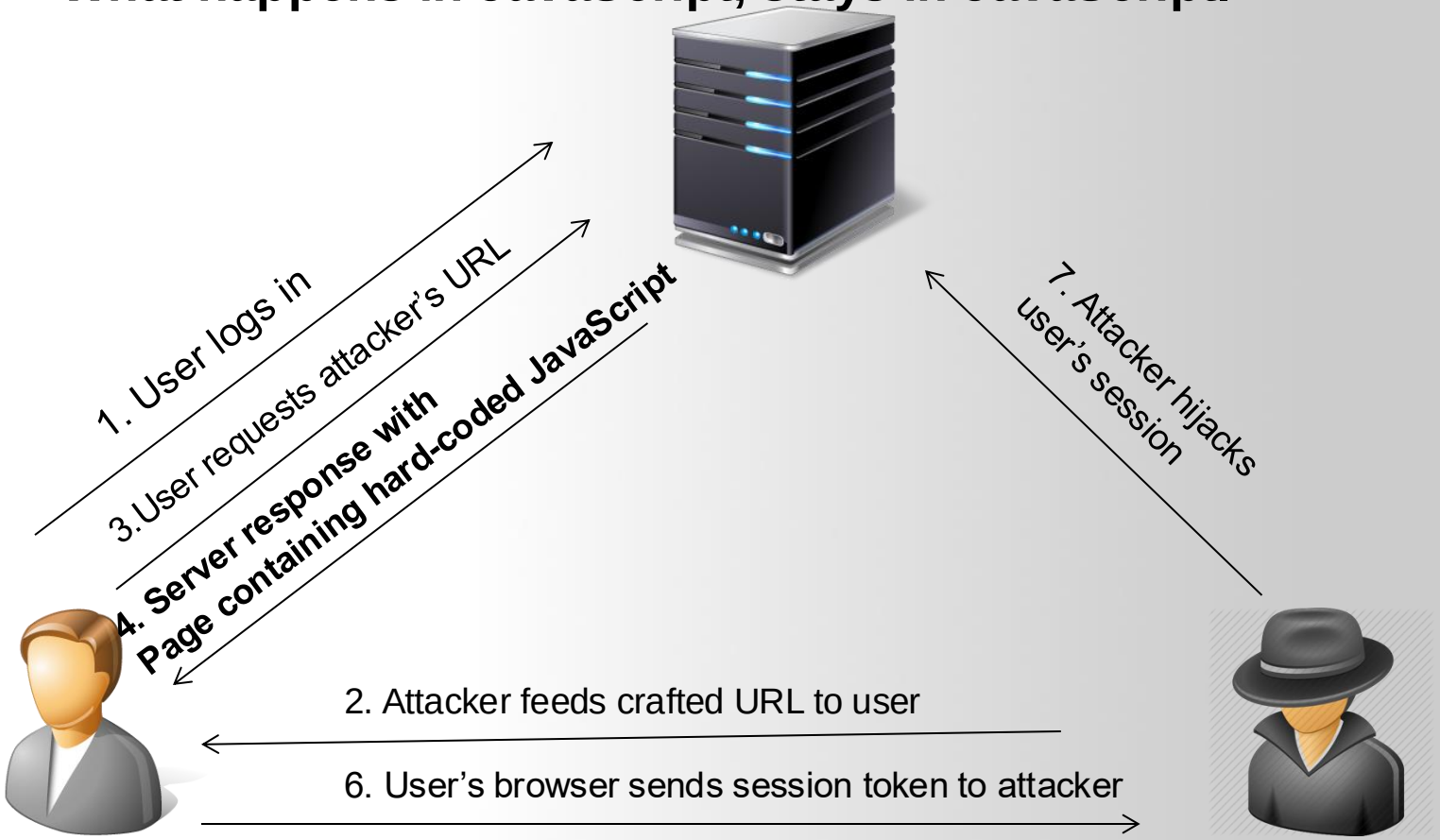
The Security Hole:

```
<?php
   $query = $pdo->prepare("UPDATE users SET first = ? WHERE id = 42");
   $query->execute(array($_POST['first_name']));
   ?>

.....

<?php
$result = $pdo->query("SELECT * FROM users WHERE id = 42"); $user = $result-
>fetchObject();
?>
<p>Welcome to <?= $user->first ?>'s Profile</p>
```

Attacking Users: Cross-Site Scripting (XSS)

XSS- DOM XSS:

What happens in JavaScript, stays in JavaScript.



Attacking Users: Cross-Site Scripting (XSS)

XSS- DOM XSS:

What happens in JavaScript, stays in JavaScript.

1. A user requests a crafted URL supplied by the attacker and containing embedded JavaScript
2. The server's response does not contain the attacker's script in any form
3. When the user's browser process this response, the script is executed nonetheless.

XSS- DOM XSS:

What happens in JavaScript, stays in JavaScript.

The Security Hole:

```
<script>
    $('#verify').submit(function() {
        var first = $(this).find("input[name=first]").val();
        $(body).append("<p>Thanks for the submission: " + first + "</p>");
        return false;
    });
    </script>
```

XSS is Everywhere

- XSS is by far the most prevalent web app vulnerability
- XSS is often misunderstood because the proof of concept (pop-up) doesn't demonstrate true attacker capability
- XSS can lead to reputational damage, denial of service, and chained exploit.
- XSS can be used against site administrators

Attacking Users: Cross-Site Scripting (XSS)

Real-world Scenario of XSS Attack

Facebook makes use of PHP scripts. The following script became vulnerable to cross-site scripting some time in July 2010:

www.facebook.com/ads/create/photos/creative_uploader.php

This script takes various parameters, one of which (controller_id) was writing user input directly inside a script tag. Take the following URL as example:

www.facebook.com/ads/create/photos/creative_uploader.php?controller_id=**c4c288b438ed080**&path=whatever&src=whatever&vol=90&w=60&h=80&post_upload=1

```
<script>
...
onloadRegister(function ()
{window.parent.__UIControllerRegistry["c4c288b438ed080"].saveUploadedImage("whatever",
"whatever", 90, 60, 80);});
...
</script>
```


https://www.acunetix.com/websitesecurity/xss-facebook/

Attacking Users: Cross-Site Scripting (XSS)

Real-world Scenario of XSS Attack

By **inserting a double quote**, an attacker is able to **escape the Array's key string** and insert JavaScript directly within a page on facebook.com.

```
<script>
...
onloadRegister(function ()
{window.parent.__UIControllerRegistry["c4c288b438ed080"].saveUploadedImage("whatever",
"whatever", 90, 60, 80);});
...
</script>
```

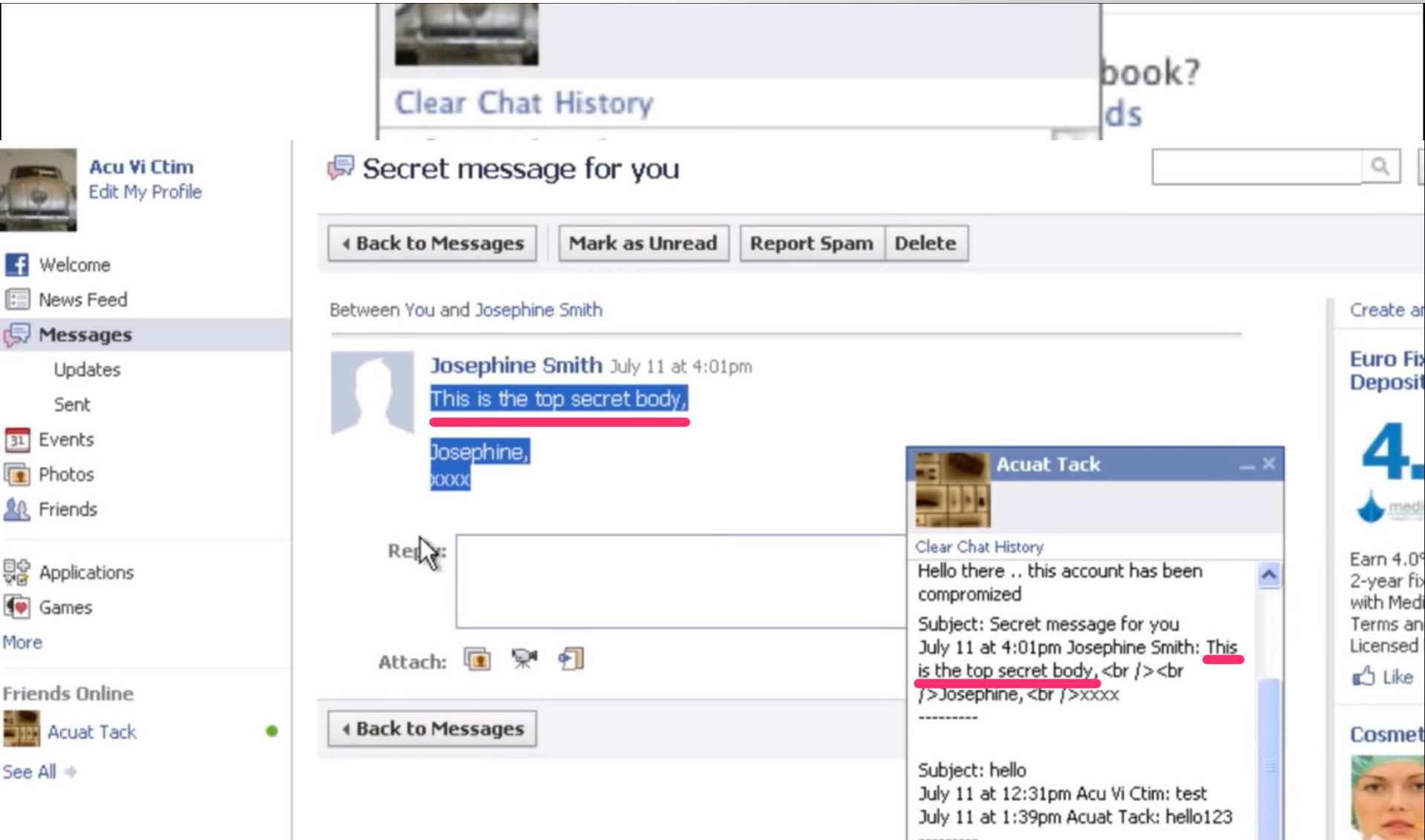
```
controller_id=test"]}; alert("facebook test"); //
```

```
<script>
...
onloadRegister(function (){window.parent.__UIControllerRegistry["test"]}; alert("facebook
test"); //"].saveUploadedImage("whatever", "whatever", 90, 60, 80);});
...
</script>
```

Attacking Users: Cross-Site Scripting (XSS)

Real-world Scenario of XSS Attack

https://www.acunetix.com/websitesecurity/xss-facebook/



Why XSS detection is hard

- Extremely difficult to automate tests for XSS
- Often times XSS defense can be bypassed in clever ways
- Developers should strive to use 3rd party libraries that are collaboratively maintained

Attacking Users: CSRF (Cross Site Request Forgery)

- In cross-site request forgery (CSRF) attacks, the attacker creates an *innocuous-looking* website that causes the user's browser to submit a request **directly to the vulnerable application** to perform some unintended action that is beneficial to the attacker.
- Normally, "the same-origin" policy does not prohibit one website from issuing requests to a different domain.

```
POST /auth/390/NewUserStep2.ashx HTTP/1.1
Host: quake0day.com
Cookie: SessionId=89BE912093001AB23B907
Content-Type: application/x-www-form-urlencoded
Content-Length: 83

realname=hackersichen&username=hacker&userrole=admin&password=12345&confirmpassword=12345
```

1. The request performs a privileged action. In the example shown, the request creates a new user with administrative privileges
2. The application relies solely on HTTP cookies for tracking sessions. No session-related tokens are transmitted else where within the request
3. The attacker can determine all the parameters required to perform the action. Aside from the session token in the cookie, no unpredictable values need to be included in the request.

Attacking Users: CSRF (Cross Site Request Forgery)

An attacker can construct a webpage that makes a cross-domain request to the vulnerable application containing everything needed to perform the privileged action.
Here is an example of such an attack.

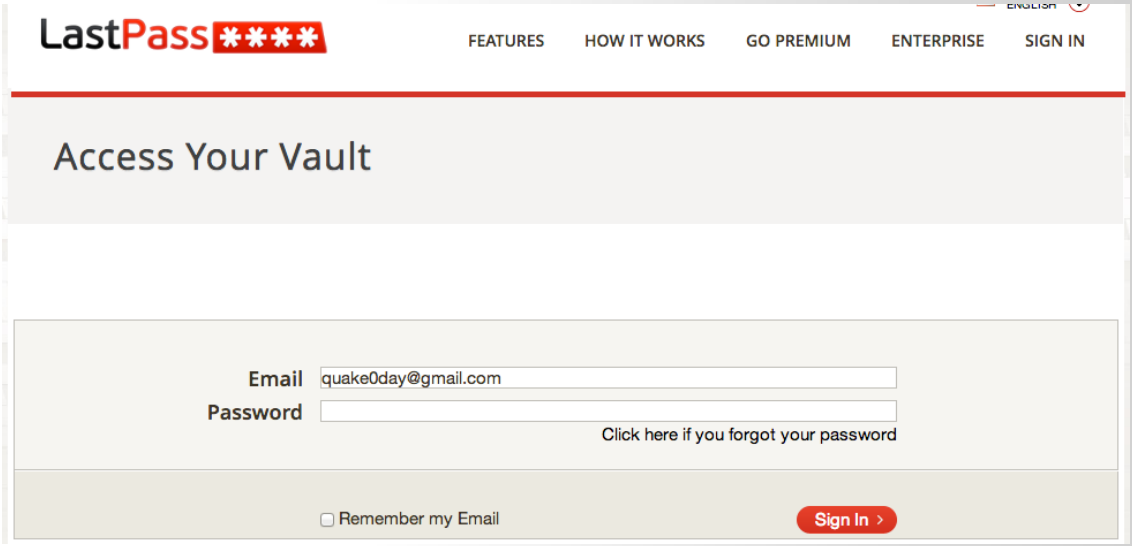
```
1   <html>
2   <body>
3       <form action="https://quake0day.com/NewUserStep2.ashx" method="POST">
4           <input type="hidden" name="realname" value="hackersichen">
5           <input type="hidden" name="username" value="hacker">
6           <input type="hidden" name="userrole" value="admin">
7           <input type="hidden" name="password" value="12345">
8           <input type="hidden" name="confirmpassword" value="12345">
9       </form>
10      <script type="text/javascript">
11          document.forms[0].submit();
12      </script>
13  </body>
14  </html>
```

This form will be automatically submitted. When the user's browser submits the form, **it automatically adds the user's cookies** for the target domain. If an admin user who is logged in the vulnerable app visits this web page, the requests is **processed within the administrator's session**.

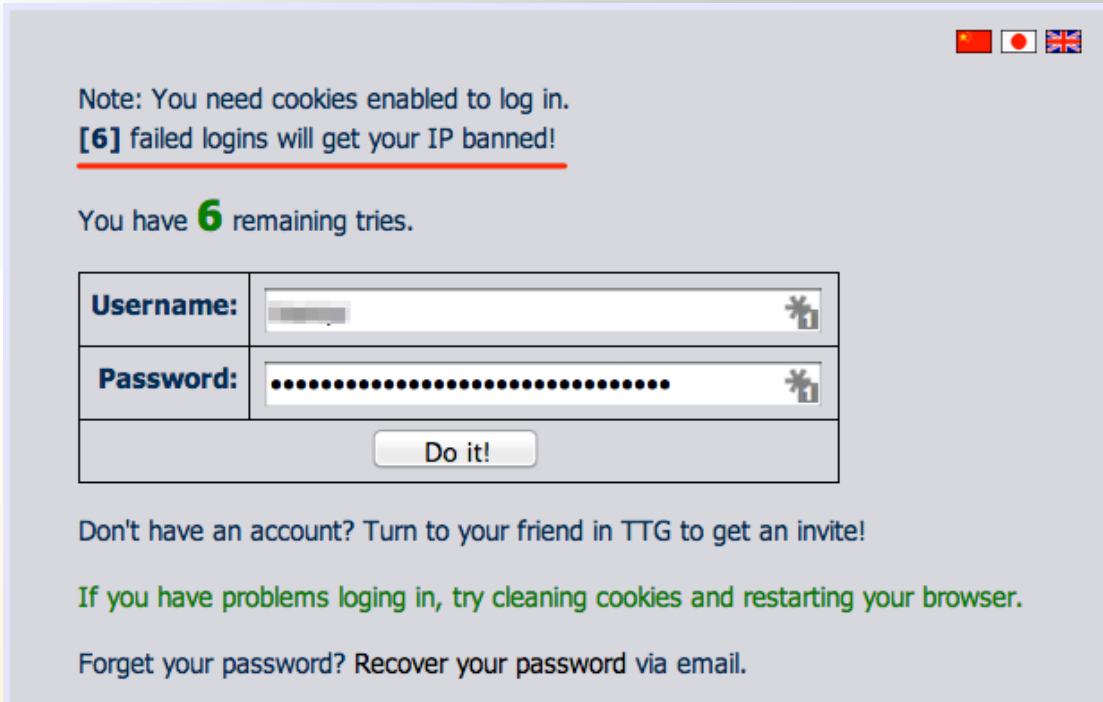
Attacking Users: CSRF (Cross Site Request Forgery)

- Cross site request forgery is a trust exploit
- The server trusts (wrongly) the browser request of users b/c authentication cookies are supplied
- Attacker forces the users browser to take action on their behalf
- Clever way to take over web applications

Brute Force Attacks (Password)



- **CAPTCHA**
- **IP rate limiting**



Q & A