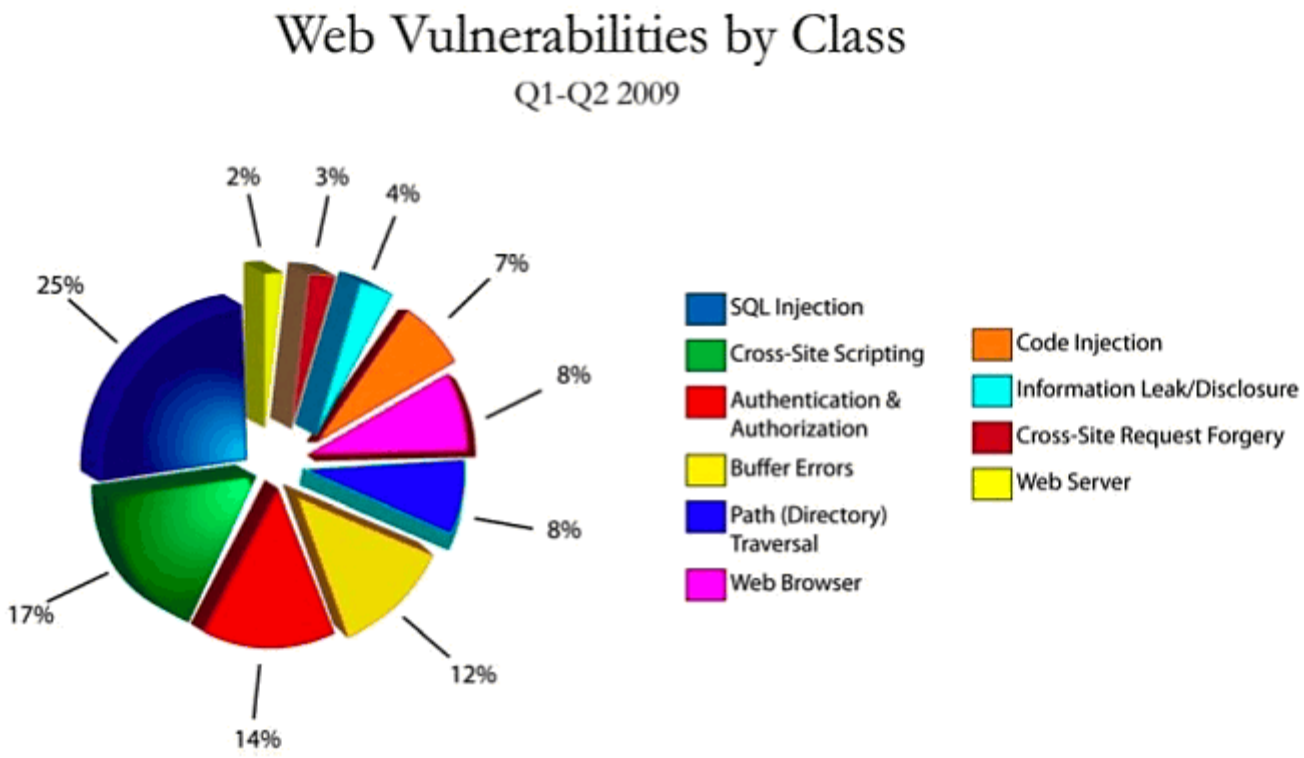
Web Security – Part 3: SQL Injection

Haipeng Dai

haipengdai@nju.edu.cn
313 CS Building
Department of Computer Science and Technology
Nanjing University

SQL Injection and XSS are top 2 attacks



Source: http://media.smashingmagazine.com/cdn_smash/wp-content/uploads/2010/01/4239939571_b7d3cddc83_o.gif

Normal SQL Queries on Web

- Most web applications involve database queries.

Phonebook Record Manager

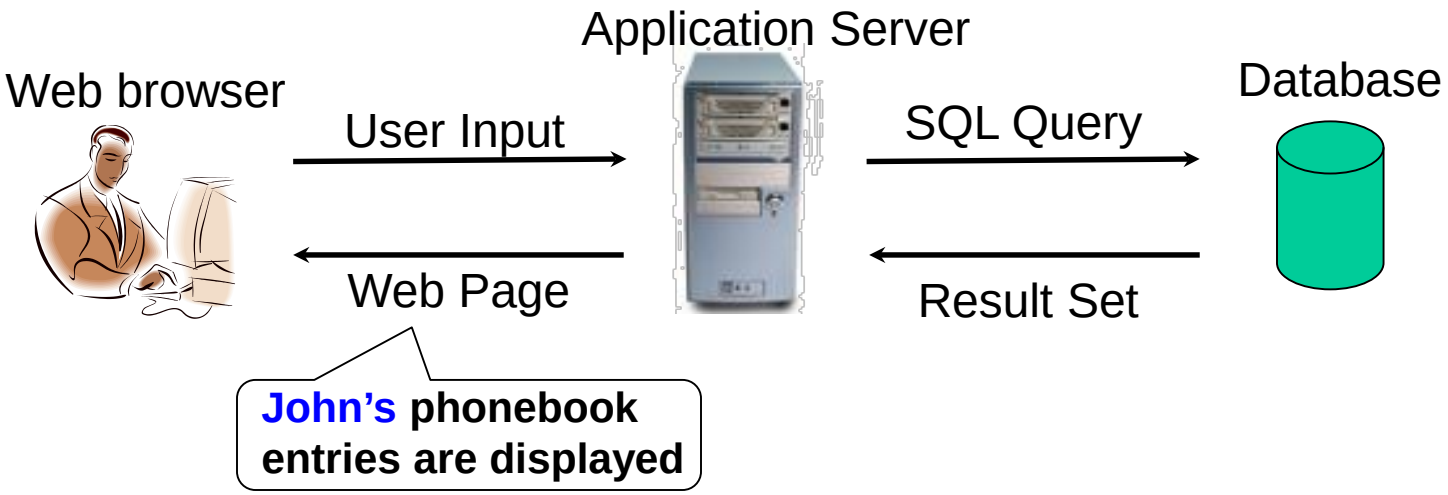
Username John

Password abcd

Submit

SQL: Structured Query Language. Used for query, delete, insert, and update database records.

SELECT * FROM phonebook WHERE username = 'John' AND password = `abcd'



SQL Injection

- Malicious query input:

Phonebook Record Manager

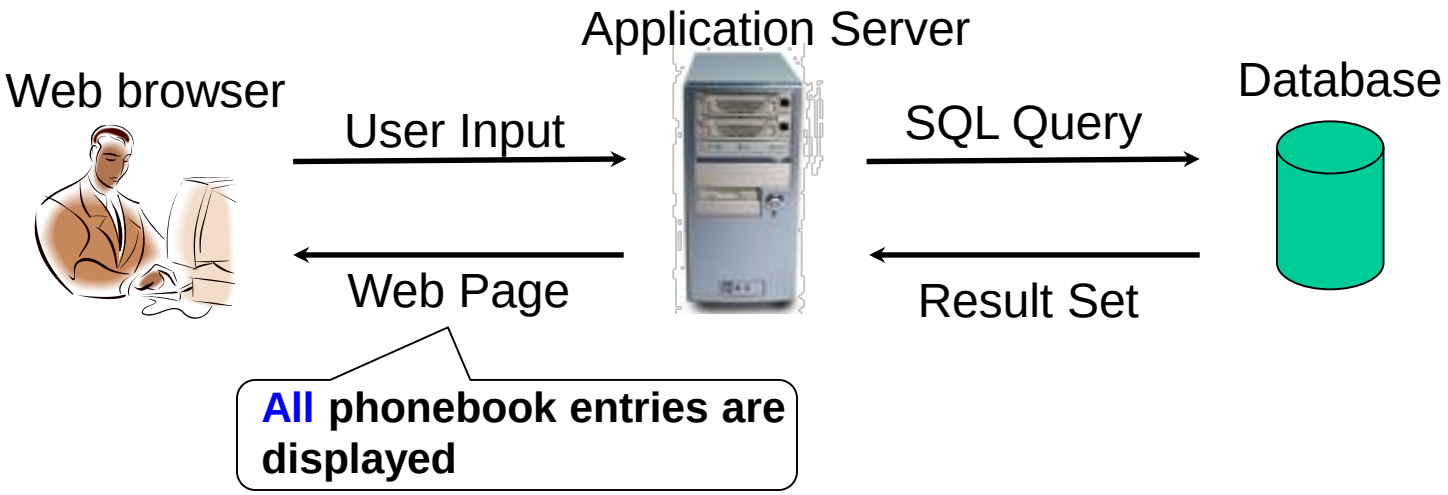
Username: John ' OR 1=1 --

Password: not needed

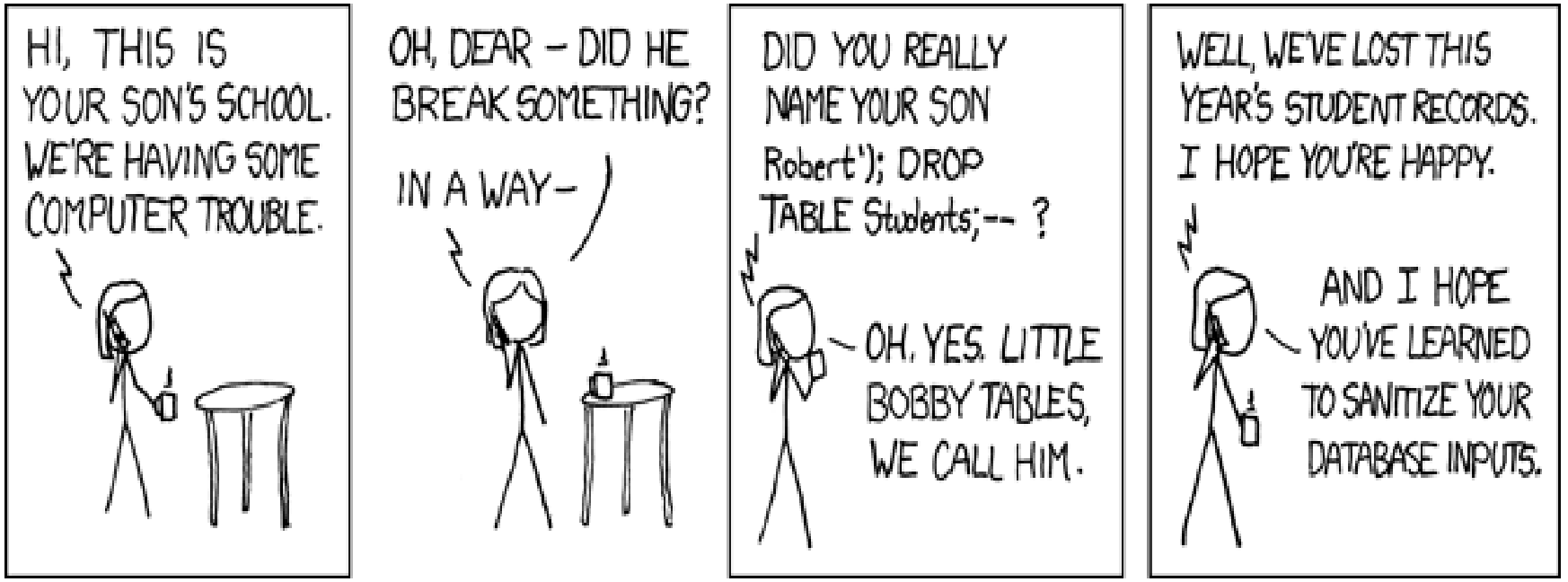
Submit

SELECT * FROM phonebook WHERE username = 'John' OR 1=1 --' AND password = 'not needed'

Everything after -- is ignored!

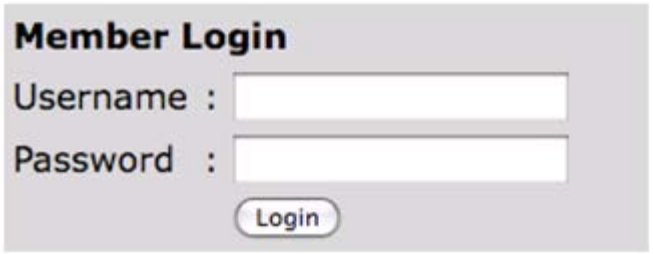


Exploits of a mum



Source: http://imgs.xkcd.com/comics/exploits_of_a_mom.png

Another SQL Injection Example (1/2)



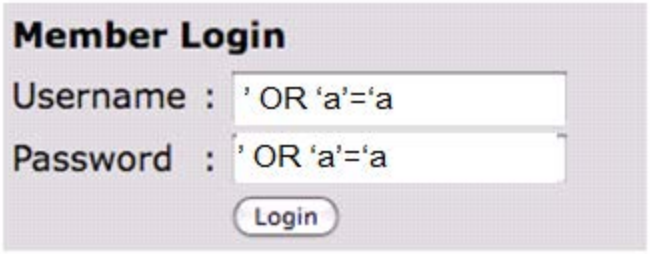

```php
<?

function connect_to_db(){...}
function display_form(){...}
function grant_access(){...}
function deny_access(){...}

connect_to_db();

if (!isset($_POST['submit'])) {
    display_form();
}
else{
    // Get Form Data
    $user = stripslashes($_POST["username"]);
    $pass = stripslashes($_POST["password"]);

    // Run Query
    $query = "SELECT * FROM `login` WHERE `user`='$user' AND `pass`='$pass'";
    echo $query . "<br><br>";
    $SQL = mysql_query($query);

    // If user / pass combo found, grant access
    if(mysql_num_rows($SQL) > 0)
    grant_access();

    // Otherwise deny access
    else
    deny_access();
}

?>
```

Another SQL Injection Example (2/2)

- SQL injection for querying data:

  SELECT * FROM `login`

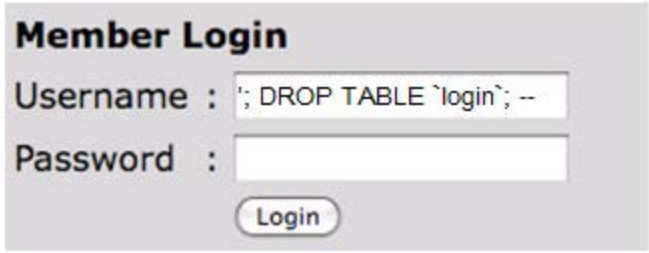
  WHERE `user`='' OR 'a'='a' AND `pass`='' OR 'a'='a'



- SQL injection for deleting data:

  SELECT * FROM `login`

  WHERE `user`=''; DROP TABLE `login`; --' AND `pass`=''



All Queries are Possible in SQL Injection

Insert record:

```
SELECT * FROM `login`
WHERE `user`='';
INSERT INTO `login` ('user','pass') VALUES ('haxor','whatever');
--' AND `pass`=''
```

Update record:

```
SELECT * FROM `login`
WHERE `user`='';
UPDATE `login` SET `pass`='pass123' WHERE `user`='timbo317';
--' AND `pass`=''
```

SQL Injection

- Insertion of SQL statements into application inputs to corrupt, exploit, or otherwise damage an application database.
- Most commonly done directly through web forms, but can be directed through URL hacking, request hacking using debugging tools, or using bots that emulate browsers and manipulate web requests.

SQL Injection in Real-life (1/2)

- On October 31, 2004, After being linked from Slashdot, the Dremel site was changed to a Goatse pumpkin
- On October 26, 2005, Unknown Heise readers replaced a page by the German TV station ARD which advertised a pro-RIAA sitcom with Goatse using SQL injection
- On January 13, 2006, Russian hackers broke into a Rhode Island government web site and allegedly stole credit card data from individuals who have done business online with state agencies.
- On November 01, 2005, A high school student used SQL injection to break into the site of a Taiwanese information security mazagine from the Tech Target group and steal customer's information.

SQL Injection in Real-life (2/2)

- On March 29, 2006, Susam Pal discovered an SQL injection flaw in www.incredibleindia.org, an official Indian government tourism site.
- On January 1, 2007, Dr.Jr7 SQL injected Nokia's website in a rather tame and civil way, but then Digg users proceeded to change it to Goatse and bukkake
- On March 2, 2007, Sebastian Bauer discovered an SQL injection flaw in knorr.de login page.
- On June 29, 2007, Hacker Defaces Microsoft U.K. Web Page using SQL injection.
- On August 12, 2007, The United Nations web site was defaced using SQL injection.

SQL Injection Prevention

- Design Principles:
  - Avoiding application structures that leave apps vulnerable
- Coding Practices:
  - Preventing bad SQL fragments from being executed
  - Blocking bad input/input sanitation
- Database Practices:
  - Making the database less vulnerable to any type of attack
- Infrastructure Support:
  - Preventing attacks on any application

Design Principle-No Anonymous User Input Data

- Force users to create an account, which is verified with an email.
- Use Captcha or similar graphics to text entry to prevent automated/bot data entry into systems.
- Log all data entry by web request – who, what, where, when and from which IP.

Design Principle - Authentication

- Eliminate all database based usernames and passwords stores.
  - A login page is the entry point into an application and must allow anonymous data entry.
  - SQL injection is frequently used to bypass security.
- Many inexpensive and free alternatives exist for authentication stores
  - OpenLDAP is easy, free, and access is through LDAP calls and not SQL.

Caution: Do not mix internal and external users in the same LDAP store if possible.

Design Principle- Avoid free text where possible and never accept HTML tags

- Constraining inputs to drop downs and formatted text boxes simplifies validations necessary to trap SQL injection attempts
- HTML tags are a very common malware vector.
  - Better to break up input into multiple text fields.
  - Use formatting options through drop downs, check boxes and other fixed input fields.

Coding Practice – Strong type checking before interacting with the database

- On the server, a request processor must perform strong type checking:
  - Ensure numbers are numbers, dates are dates, values from form elements are correct such as indexes from drop downs, etc.
  - Limit the range of values accepted if possible.
  - Use the parsing functions that come natively with many programming languages if available such as in Java or .NET
  - It is very important in weakly typed languages such as PHP to force type checking.

Coding Practice – Enforce input lengths and formats

- Limit the size of all strings on both the client and the server.
  - Reject any request where any value exceeds a maximum expected length.
  - Sometimes this causes bad usability. For example, an address box cannot hold a long address.

Caution: Rules must be implemented on the client and on the server because rules implemented on the client may be bypassed.

Coding Practice – Sanitize all user input before any other processing

- The safest and most secure practice is to iterate through a web request and filter all unexpected characters.
  - If all special characters are removed, function calls, URI encoding, and other common ways of adding SQL predicates or embedded HTML tags are simply blocked.
- Reject requests with anomalies and log the activity for analysis.

Coding Practice – Mask all errors from the user with user friendly output

- Never display sql errors or other raw system errors back to the user.
  - Can provide additional attack vectors for hackers.
- Whenever an exception occurs, display a generic message, and log the actual error and user input.
- Whenever a request fails validation or sanitation checks, use a generic response, terminate the user's session, and log the error in detail.

Caution: ***NEVER echo user input back to the user without sanitizing the request. This is the most common form of cross site scripting.***

Coding Practice – Use Detailed Logs

- Detailed logging is useful!
  - It introduces additional storage and process overhead, but it is invaluable in debugging and in identifying security weaknesses.
- Unexpected conditions, rejected requests, and similar errors are usually the first sign your web application is under attack.

Coding Practices – Use frameworks

- Every popular web development platform has validation frameworks.
- Leverage existing frameworks to implement validations:
  - Zend for PHP
  - Several frameworks for Java
- Frameworks can centralize many security related tasks.

Database Practice – Use two accounts

- Create Two accounts:
  - Database Owner
    - Has rights over all the objects in a database or schema.
    - Equivalent to DBA level access for a database/schema.
    - Used to build out and maintain an application database.
    - Never used by web applications.
  - Application Account/Database proxy account
    - Has minimal rights needed for application:
      - All rights to each object are explicitly declared.
      - Owns no objects directly.
      - No access to metadata in db platform.
      - Restricted login locations if possible.

Database Practices – Strong Typing

- Columns must be strongly typed:
  - Numbers as Numbers.
  - Characters limited to the exact maximum required.
  - Dates stored as dates
- If performance is acceptable use check constraints or triggers:
  - Force format masks and character ranges such as 0-9 for SSN, etc.

Database Practices – Stored Procedures and views

- Views:
  - Only expose those columns needed by the application.
  - Allow for more granular column by column permissions.
- Stored Procedures:
  - Application account get execution rights only.
    - All tables and views are invisible.
  - Can reduce number of database interactions.
  - Simplifies transaction management.
  - Not appropriate for all application environments/tools.

Database Practices – Configure database error reporting

- Default error reporting often gives away information that is valuable for attackers (table name, field name, etc.)
- Configure so that this information is never exposed to a user

Infrastructure Practice – Deploy an IDS specifically checking for SQL Injection

- Several IDS systems exist to specifically monitor web traffic for SQL Injection
  - Each request is examined for SQL injection signatures.
  - Bad requests are filtered and logged.
- Protects all applications against most common errors.
- Excellent first step until all web applications can be reviewed for vulnerabilities.

Caution: Signatures will always eventually be defeated.

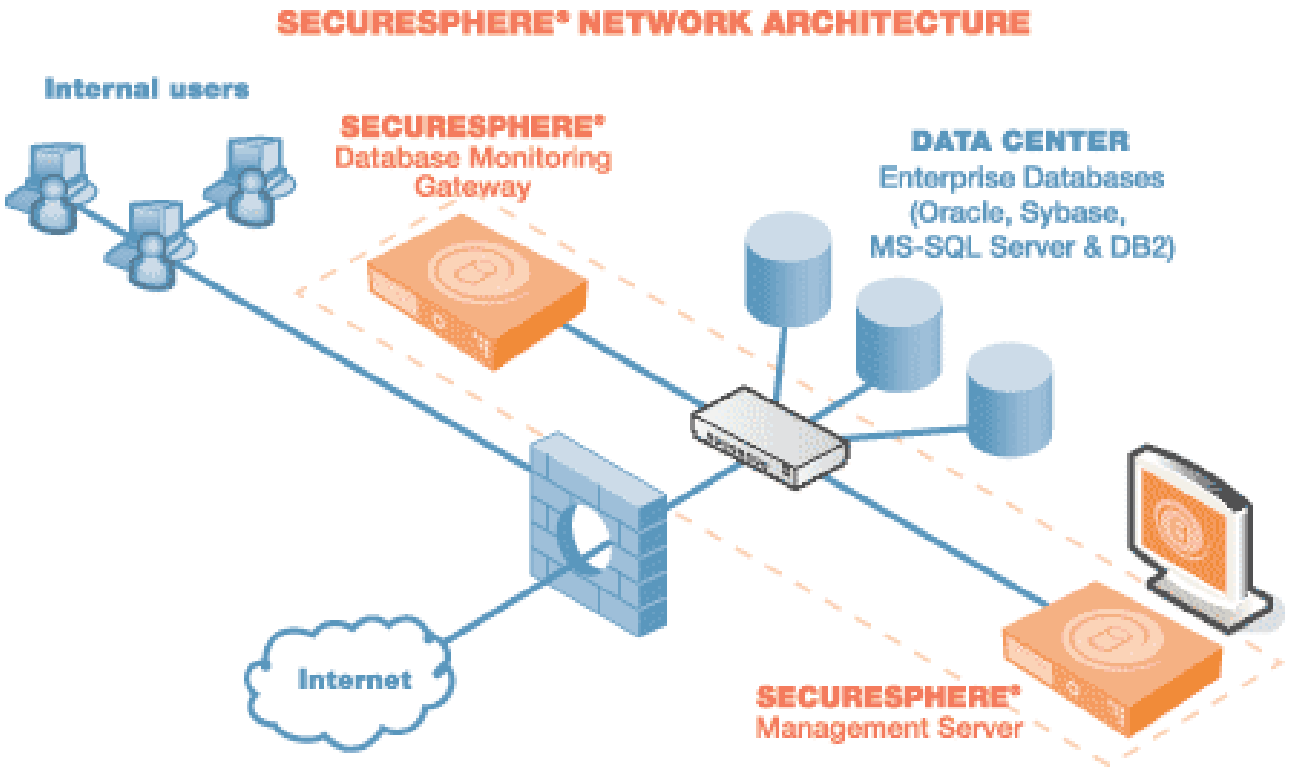
Infrastructure Practice – Do automated log scanning

- Use a central logging tool looking for SQL Injection behavior:
  - SQL Errors
  - Queries against metadata from web apps
- Use manual scanning tools such as grep, awk, and log parsers to look for unauthorized queries/sql requests

Infrastructure Practice – Use security scanning tools

- The best security measure is one that catches problems before they are revealed.
- Applications should be automatically scanned for vulnerabilities.
- Reveals vulnerabilities above and beyond simple penetration testing.
- Many excellent products:
  - Rational APPSCAN from IBM (was Watchfire)
  - Acunetix

Infrastructure Practice – Example



http://www.imperva.com/products/securesphere/

Performance Metric

SecureSphere Throughput up to 2 Gbps .

SQL Requests/sec up to 200,000

Other Injections

- http://victim.com/copy.php?name=username
- copy.php file includes

  system(“cp temp.dat $name.dat”)
- User calls

  http://victim.com/copy.php?name=“a; rm *”
- copy.php executes

  system(“cp temp.dat a.dat; rm *”);

Web Security – Part 4: Cross-Site Request Forgery

Alex X. Liu & Haipeng Dai

haipengdai@nju.edu.cn
313 CS Building
Department of Computer Science and Technology
Nanjing University

Cross-Site Request Forgery (XSRF)

- 1. You log in https://www.bank.com. Do some bank stuff. But you have not logged off yet.
- 2. You open another page www.attacker.com. This page has an evil form:

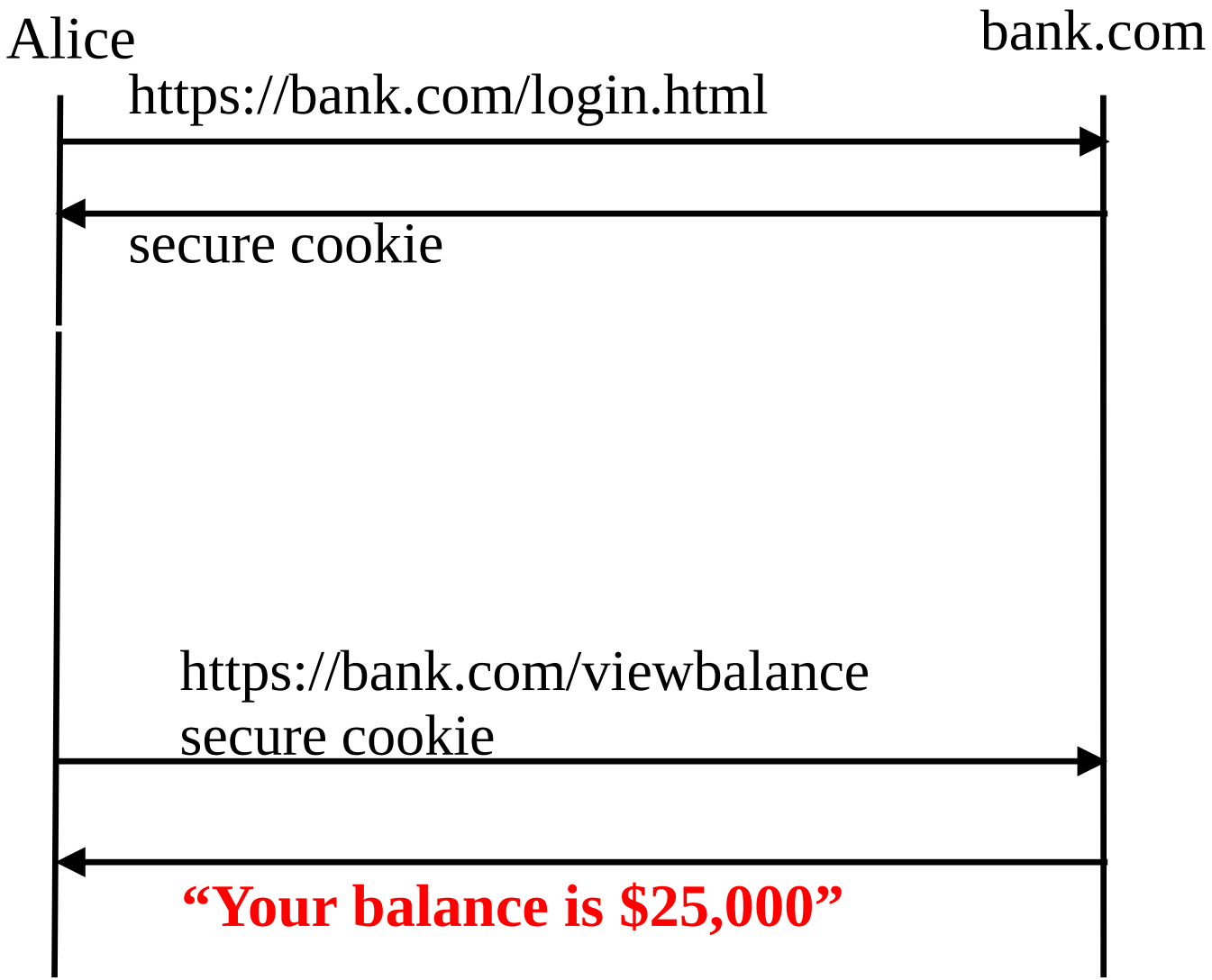
```
<form method="POST" name="evilform" target="hiddenframe"
  action="https://www.bank.com/update_profile">
  <input type="hidden" id="password" value="evilhax0r">
</form>
<iframe name="hiddenframe" style="display: none"> </iframe>
<script>document.evilform.submit();</script>
```

- 3. Your browser executes this form. The result is that the evilform is submitted with a password-change request to bank.com's "good" form: www.bank.com/update_profile with a <input type="password" id="password"> field.
- You noticed nothing because nothing is displayed.

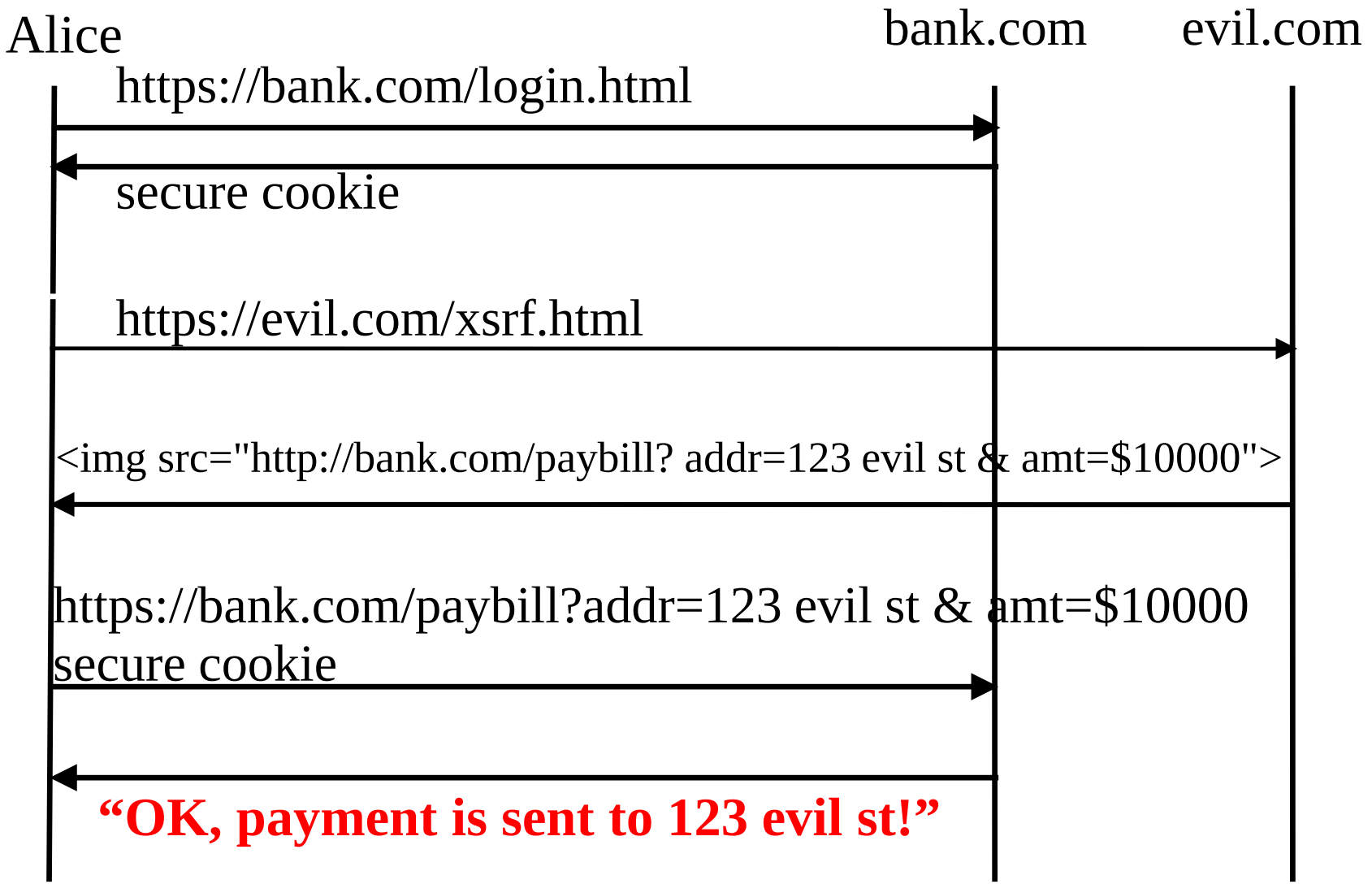
Cross-Site Request Forgery (XSRF)

- Remember that the cookie of a site is sent to the site each time the browser accesses the site.
- Also remember that the cookie is used as an authentication token.
- Attacker: what if I force your browser to make my request to the secure site with your cookie?
- Forces user to send unauthorized requests by interacting with a malicious website.
- Can force someone to transfer money, change status on social networking site, buy stock, or any other action on a vulnerable website an attacker would like to exploit.
- XSRF impacts: Malicious site can’t read bank info, but can request bank to do things benefit attackers – such as transfer money to them!

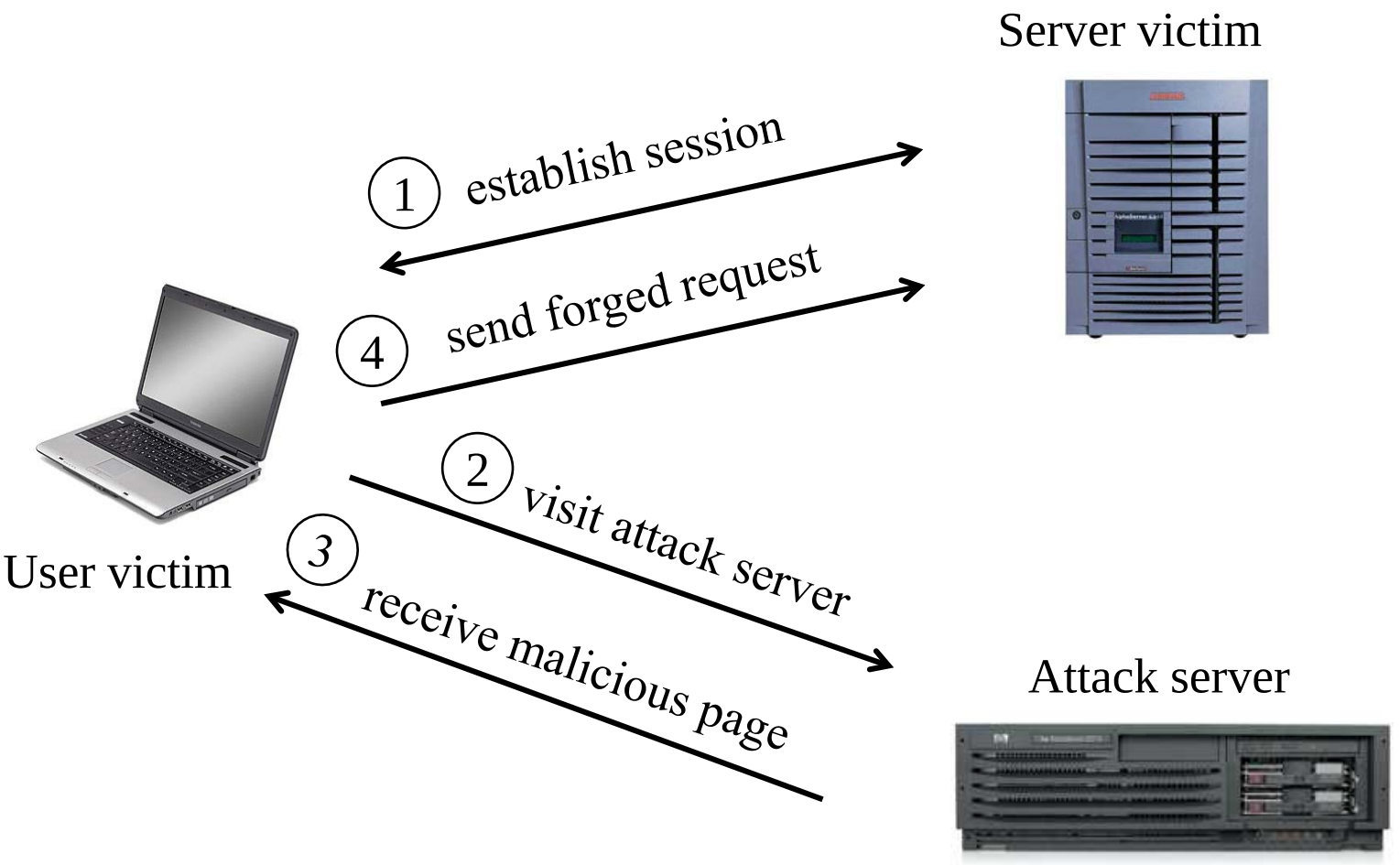
Normal Interaction



XSRF Attack


Alice
bank.com
evil.com
https://bank.com/login.html
secure cookie
https://evil.com/xsrf.html
<img src="http://bank.com/paybill? addr=123 evil st & amt=$10000">
https://bank.com/paybill?addr=123 evil st & amt=$10000
secure cookie
“OK, payment is sent to 123 evil st!”

XSRF (aka CSRF): Basic Idea



Q: how long do you stay logged on to Gmail?

Cookie Authentication is NOT Enough!

- Users logs into bank.com, forgets to sign off
  - Session cookie remains in browser state
- User then visits a malicious website containing

```
<form  name=BillPayForm action=http://bank.com/BillPay.php>
<input  name=recipient  value=badguy> …
<script> document.BillPayForm.submit(); </script>
```

- Browser sends cookie, payment request fulfilled!
- Lesson: cookie authentication is not sufficient when side effects can happen

XSRF vs. XSS

- Cross-site scripting
  - User trusts a badly implemented website
  - Attacker injects a script into the trusted website
  - User's browser executes attacker's script
- Cross-site request forgery
  - A badly implemented website trusts the user
  - Attacker tricks user's browser into issuing requests
  - Website executes attacker's requests

XSRF in Real Life

- Hi5.com—Yahoo's Social Networking Website
  - Change Profile Skin
  - Change Status
  - Add Applications
- Sharebuilder.com—ING's Online Stock Brokerage
  - Buy/Sell shares of stock
  - Requires 2 requests for attack

CSRF Defenses

- Secret Validation Token


```
<input type=hidden value=23a3af01b>
```

- Referrer Validation


```
Referrer: http://www.facebook.com/home.php
```

- Custom HTTP Header


```
X-Requested-By: XMLHttpRequest
```

Secret Validation Token vs. Web Attacker

- Hash of User ID
  - Attacker can forge

```
<input type=hidden value=23a3af01b>
```

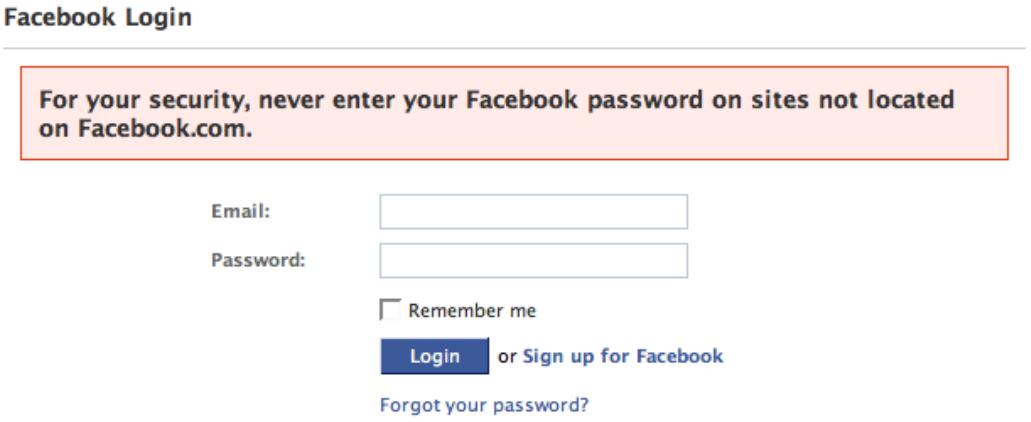
- Session-Dependent Nonce (CSRFx, CSRFGuard)
  - Requires managing a state table
- **HMAC of Session ID**
  - **No extra state required**

Referrer Validation

Referrer: http://www.facebook.com/home.php

Referrer: http://www.evil.com/XSRFattack.html

Referrer:



- Lenient Referrer checking – header is optional

Referrer:

- Strict Referrer checking – header is required

Referrer:

Why use Lenient Referer Checking?

- Referer may leak privacy-sensitive information

  `http://intranet.corp.apple.com/projects/iphone/competitors.html`

- Common sources of blocking:
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS -> HTTP transitions
  - User preference in browser
  - Buggy user agents
- Site cannot afford to block these users

Lenient Referrer Checking is not secure! Don't use it!

Standford Proposal: Origin Header

- Privacy
  - Identifies only principal that initiated the request (not path or query)
  - Sent only for POST requests; following hyperlink reveals nothing

```
Origin: http://www.evil.com
```

- Usability
  - Authorize subdomains and affiliate sites with simple firewall rule

```
SecRule REQUEST_HEADERS:Host !^www\.example\.com(:\d+)?$ deny,status:403
SecRule REQUEST_METHOD ^POST$ chain,deny,status:403
SecRule REQUEST_HEADERS:Origin !^(https?://www\.example\.com(:\d+)?)?$
```

  - No need to manage secret token state
  - Can use redundantly with existing defenses to support legacy browsers
- Standardization
  - Supported by W3C XHR2 and JSONRequest
  - Expected in IE8's XDomainRequest