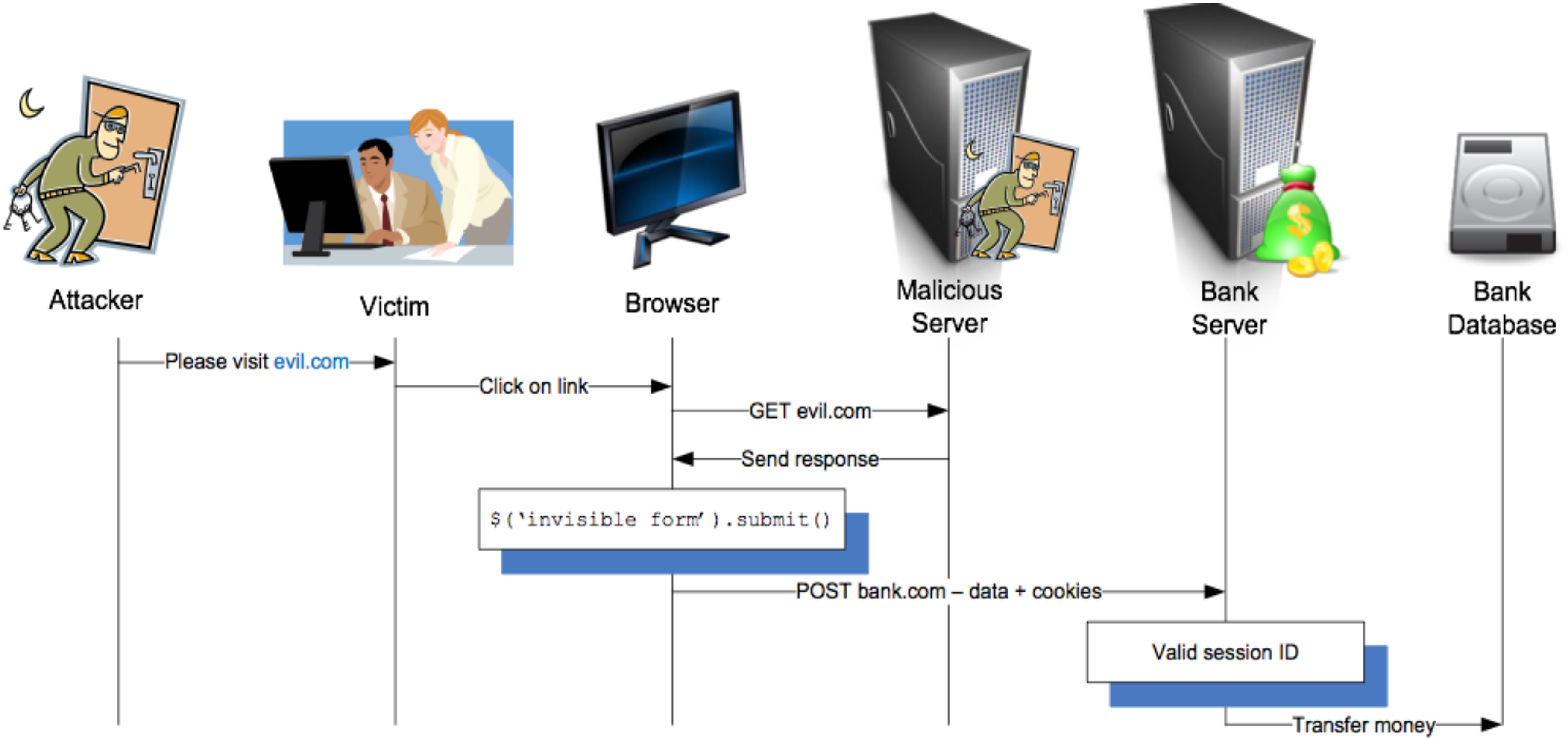
A Server and Browser-Transparent CSRF Defense for Web 2.0 Applications

Slides by Connor Schnaith

Cross-Site Request Forgery

- One-click attack, session riding
- Recorded since 2001
- Fourth out of top 25 most dangerous software errors
  - CWE/SANS
- Takes advantage of cookies
  - can send malicious requests under user credentials
  - potential to steal user money, etc.
- Relies on tricking the user into clicking a malicious link, often embedded into an image

CSRF Example



```
$('invisible form').submit()
```

Figure 1: Illustration of a CSRF attack

Why does CSRF work?

- User's browser automatically sends credential information on login

- Browsers enforce no restrictions on outgoing requests
  - SOP does not allow cookies to be viewed or written by any source other than originator
  - CSRF does not rely on tampering with cookies

Current CSRF Defenses: browsers

- Goal: ensure that every sensitive request originates from own pages
- Referrer header in http requests
  - scripts cannot alter it
  - details the originator of a request
  - too many browsers suppress it (privacy concerns)
- Origin header
  - same idea as referrer header
  - not supported by most browsers
- Browsers can't do it... up to the developers

Current CSRF Defenses: nonce

- Associating a *nonce* with each web page
  - ensure that all requests from this page provide the nonce
  - SOP prevents one domain from reading the source of another domain, so nonce cannot easily be stolen
- Adding a nonce to each page is a *manual* process
  - developer might miss a page
  - may omit because mistaken belief that a particular request is not vulnerable

Current CSRF Defenses: Products

- NoForge
  - Uses nonce approach
  - On server side, intercepts every page sent to a client
  - Re-writes URLs found found on the page so that they supply the nonce when requested
- stRodeo
  - Similar, but deployed on client-side
- Neither protects dynamic construction of web pages on the browser (web 2.0 applications) since depend on static rewriting of link names

Current CSRF Defenses: Drawbacks

- Need for programmer effort and/or server side modifications
- Incompatibility with current browsers
- Inability to protect dynamically generated results
- Lack of support for legitimate cross-origin requests
    - no natural way to extend products like noForge into the cross-origin domain

jCSRF: an introduction

- Transparently interposes communication between client and server

- Proxy jCSRF
  - avoids need for server-side changes
  - needs to deal with HTTPS compatibility, i.e. encrypted data

- Server plug-in jCSRF
  - server must support plug-in architecture (Apache)
  - less overhead than proxy

- Intercept POST but not GET requests

jCSRF: approach overview

- Step One:

  An authentication token is issued to pages served by the protected server

- Step Two:

  A request is submitted to jCSRF together with the authentication token

- Step Three:

  jCSRF uses authentication token to verify that the originator is an authorized page.

  - Validated: request is forwarded to the server
  - Not validated: request forwarded with all cookies stripped

jCSRF: javascript injection

- When page is served by protected server, javascript is automatically injected
- Also includes a new cookie in the HTTP response that can be used by the script to authenticate same-origin requests
- It is the job of the javascript to determine if the request is cross-origin or same origin

```
< script type="text/javascript" src=... > </script>
```

jCSRF: javascript injection

- Two ways in which browser may issue POST requests, which will be intercepted by jCSRF-script

- Submission of HTML forms
  - form may be dynamically generated by javascript
  - not necessary for user to submit the form, the form may be submitted automatically by javascript

- XmlHttpRequest
  - the response to a XmlHttpRequest can be read by the script making the request

HTML Form Submission

- jCSRF-script registers a submit handler for each POST-based form, determines if same or cross origin
- Same-origin
  - adds authentication token as additional parameter
- Cross-origin
  - first obtain a token for the target domain
  - adds token as additional parameter
- If the application already has its own event handler, there could be possible confusion from extra parameter
  - wraps existing handler with function that removes parameter before handler is called, and then adds the parameter after

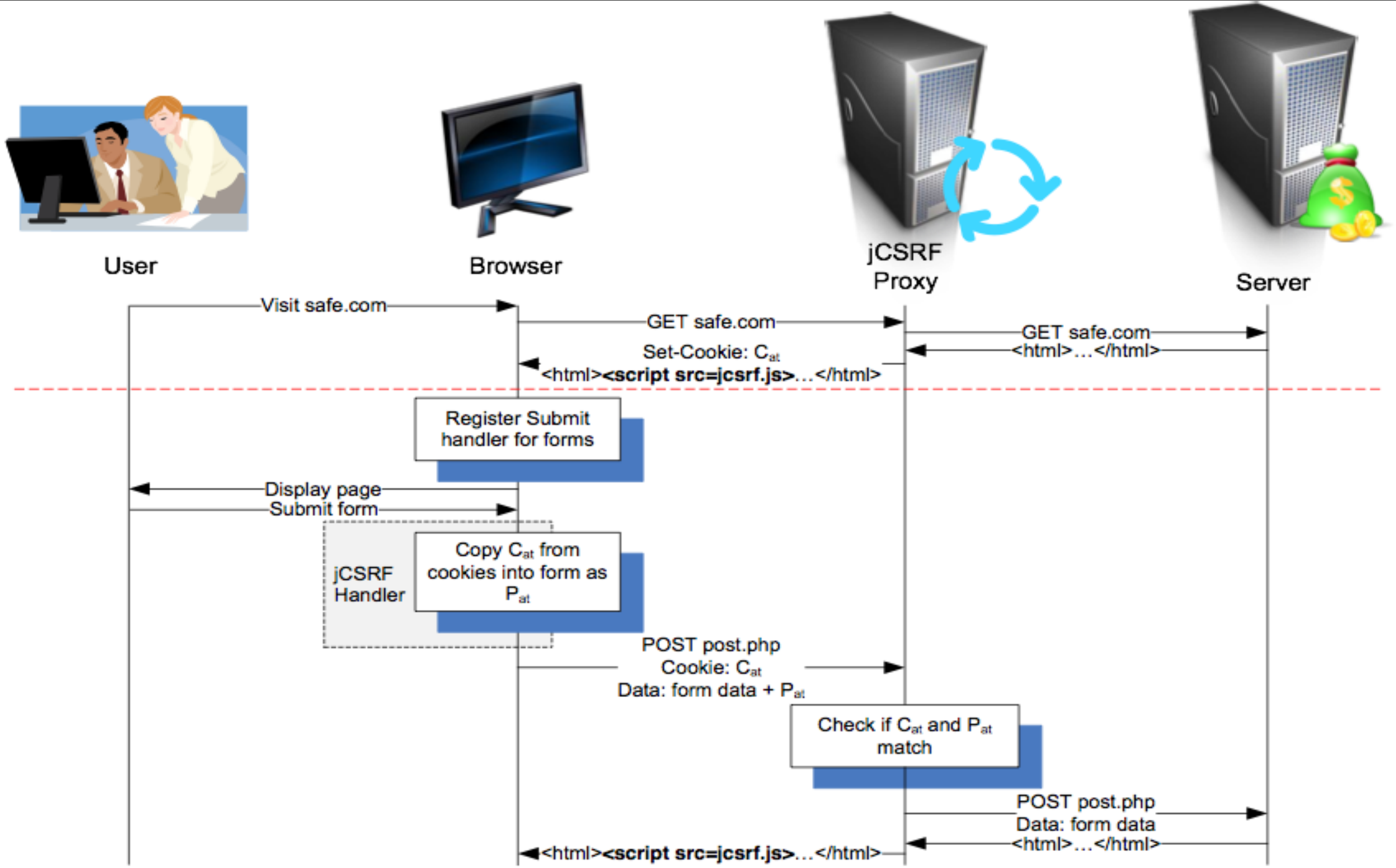
XmlHttpRequest

- jCSRF-script modifies the **send** method of the class
- If browser supports DOM prototypes, the **send** function can simply be replaced
- Older browsers the XmlHttpRequest must be wrapped in a proxy object that hides the original class and redirects all requests made by the application to the proxy class
- Adding special header X-No-CSRF, which XmlHttpRequests allow, proves that request is same-origin

jCSRF: same origin protocol

- First, the user must send a GET request
- Sets cookie, injects javascript into response
- When browser receives response, javascript executes
  - this ensures that that the value in the cookie is copied into a new parameter
- When POST is made, checks to see if cookie, *Cat*, and parameter, *Pat*, are the same
  - if attacker attempts a jCSRF, the cookie will be sent but they will not have the correct parameter in the data

jCSRF: same origin protocol



User

Browser

jCSRF Proxy

Server

Visit safe.com

GET safe.com

GET safe.com

<html>…</html>

Set-Cookie: C_{at}

<html>**<script src=jcsrf.js>**…</html>

Register Submit handler for forms

Display page

Submit form

jCSRF Handler

Copy C_{at} from cookies into form as P_{at}

POST post.php

Cookie: C_{at}

Data: form data + P_{at}

Check if C_{at} and P_{at} match

POST post.php

Data: form data

<html>…</html>

<html>**<script src=jcsrf.js>**…</html>

jCSRF: same-origin correctness

- Scripts running on an attacker-controlled page visited by users browsers cannot obtain the authentication token for the protected domain

Proof: Immediate from SOP. Since the authentication token is stored as a cookie, attackers code running on the user's browser runs on a different domain and has no access to it

jCSRF same-origin correctness

- Any token that may be obtained by the attacker cannot be used to authenticate a request from the user's to the protected domain

Proof: Again, due to SOP, the attacker cannot set a user token. Any token obtained by the attacker and embedded into forms sent by the user would not match the cookie set by jCSRF

jCSRF: same-origin correctness

- The attacker should not be able to guess an authentication token that is valid for the protected domain

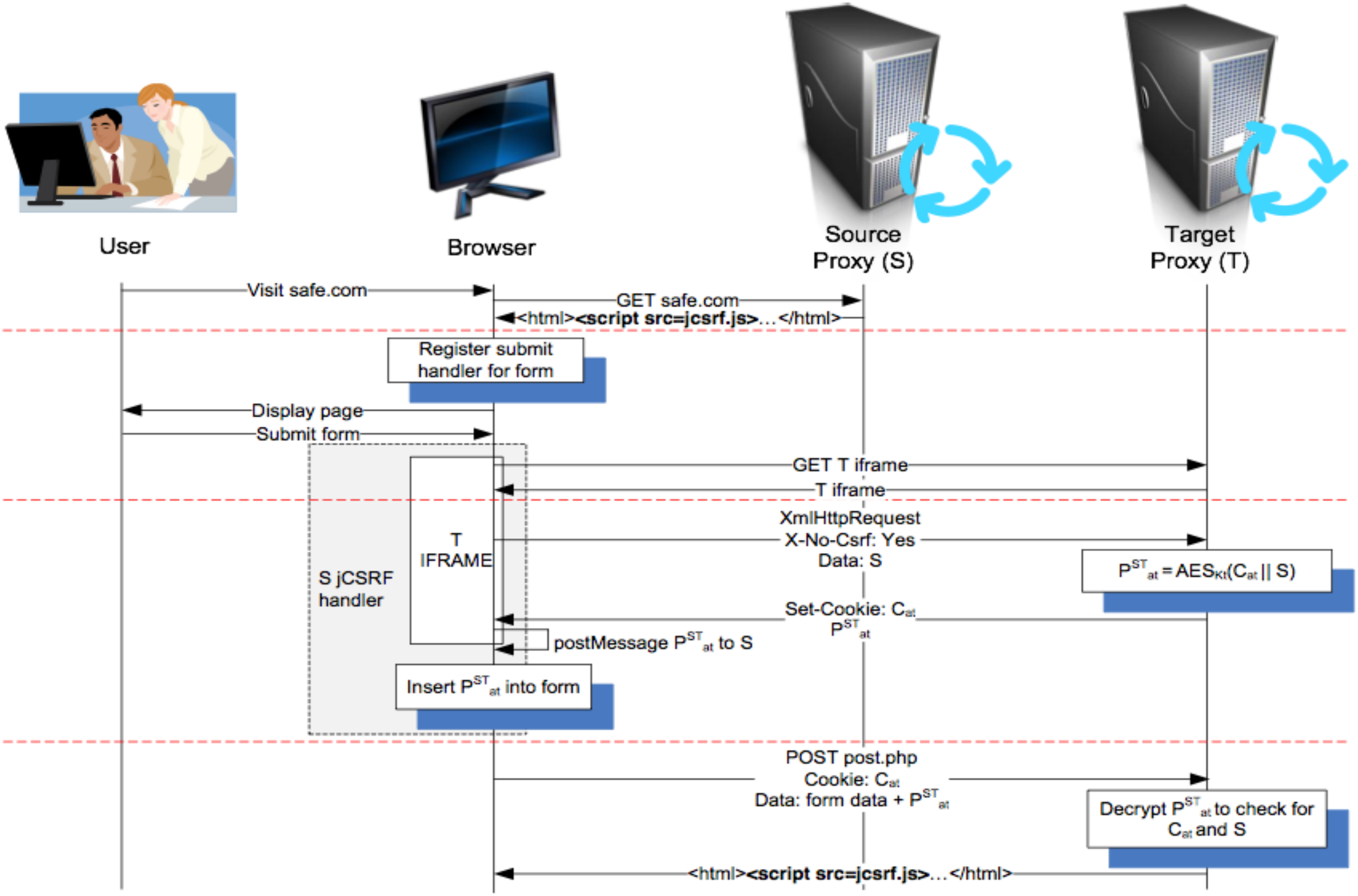
Proof: Token randomly chosen from large keyspace.
The encryption protocol is as follows:

1. A 128-bit random value *IR* is generated
2. A sequence of random numbers *R1*, *R2*, ... are generated
3. Nonces, *N1*, *N2*, ... , are generated using the following:
   Ks = *IR*, *Ni*, = *AESks(Ri)* (the AES encryption algorithm)
4. Each new C*at* it sets to *Ni* and increments i

jCSRF: cross-origin protocol

- When POST action occurs, verifies that target domain accepts requests from source domain
- Injects iframe: **http://T/jCSRF-crossdomain.html?domain=S**
  - Contains javascript that will set up token *Pat*
- XmlHttpRequest made from iframe with X-No-CSRF header
  - ensured that the request is made from same domain
- Sets cookie if not set, and PostMessage from target to source containing *Pat*
- *Pat* is added to the form and submitted
- jCSRF checks both source and token validity
  - if either fails, page requested with stripped cookies

jCSRF: cross-origin protocol



User

Browser

Source Proxy (S)

Target Proxy (T)

Visit safe.com

GET safe.com

<html>**<script src=jcsrf.js>**…</html>

Register submit handler for form

Display page

Submit form

S jCSRF handler

T IFRAME

GET T iframe

T iframe

XmlHttpRequest
X-No-Csrf: Yes
Data: S

$P^{ST}_{at} = AES_{Kt}(C_{at} \| S)$

Set-Cookie: C_{at}
P^{ST}_{at}

postMessage P^{ST}_{at} to S

Insert P^{ST}_{at} into form

POST post.php
Cookie: C_{at}
Data: form data + P^{ST}_{at}

Decrypt P^{ST}_{at} to check for C_{at} and S

<html>**<script src=jcsrf.js>**…</html>

jCSRF: cross-origin correctness

- Scripts running on an attacker-controlled page visited by users browsers cannot obtain the authentication token for the protected domain

Proof: the postMessage API only allows the attacker to receive an authentication token that includes its true domain, or it may lie about its origin and not receive a token at all

jCSRF: cross-origin correctness

- Any token that may be obtained by the attacker cannot be used to authenticate a request from the user's to the protected domain

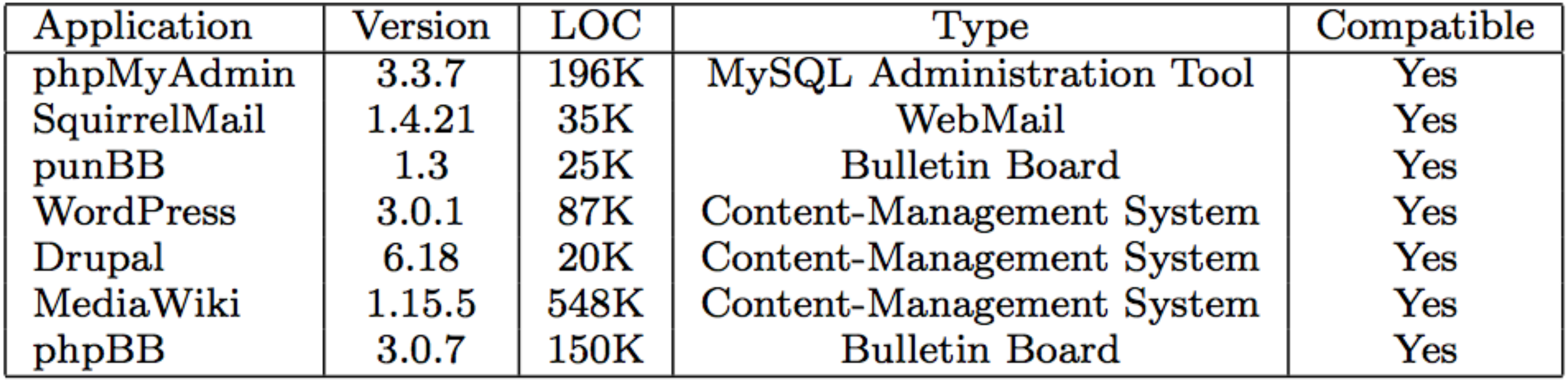
Proof: Again, due to SOP, the attacker cannot set a user token. Any token obtained by the attacker and embedded into forms sent by the user would not match the cookie set by jCSRF

jCSRF: cross-origin correctness

- The attacker should not be able to guess an authentication token that is valid for the protected domain

Proof: Cross-origin uses same encryption method as same-origin.

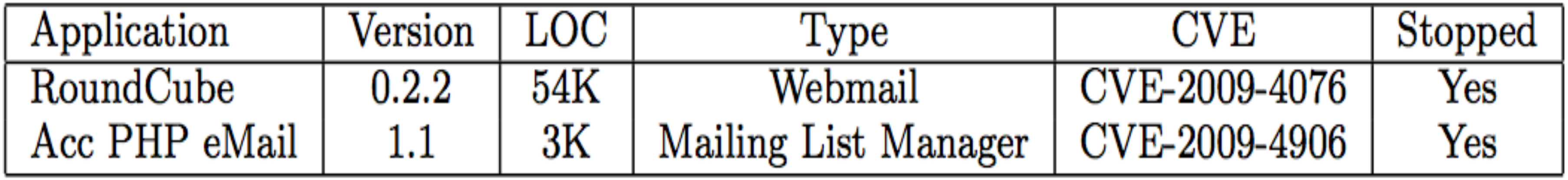
jCSRF: compatability



Application	Version	LOC	Type	Compatible
phpMyAdmin	3.3.7	196K	MySQL Administration Tool	Yes
SquirrelMail	1.4.21	35K	WebMail	Yes
punBB	1.3	25K	Bulletin Board	Yes
WordPress	3.0.1	87K	Content-Management System	Yes
Drupal	6.18	20K	Content-Management System	Yes
MediaWiki	1.15.5	548K	Content-Management System	Yes
phpBB	3.0.7	150K	Bulletin Board	Yes

- Used Firefox and Chrome
- Applications chosen for complexity and difficulty for manual CSRF protection
- Did not test for cross-origin requests

jCSRF: protection



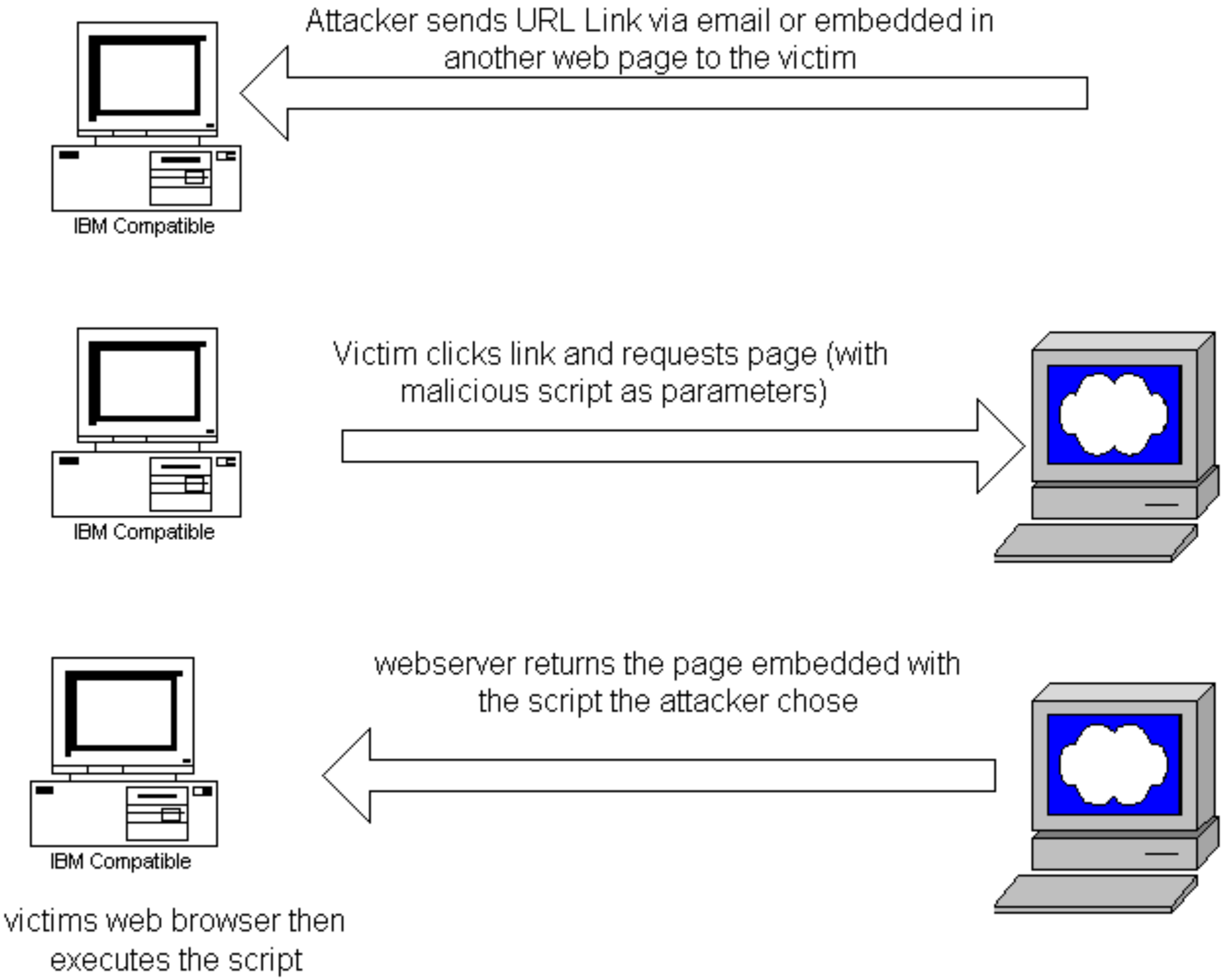
Application	Version	LOC	Type	CVE	Stopped
RoundCube	0.2.2	54K	Webmail	CVE-2009-4076	Yes
Acc PHP eMail	1.1	3K	Mailing List Manager	CVE-2009-4906	Yes

- Two known CVE vulnerabilities were exploited
- First, a fake email was posted using using RoundCube, which failed due to lack of authentication token
- Second, a message was posted to change the admin password, but the attack was thwarted
- Same-origin CSRF attacks can be successful if a form is injected into a server response
    - jCSRF will not know it is malicious and will supply it with the correct authentication token

XSS: a side note

- Break the assumption that same-origin scripts are under the control of the web developer
  - issue token requests and leak results to the attacker
- jCSRF has no way to protect against this
  - attacker can steal the cookies directly and pose as the victim from his own machine
- No CSRF defense mechanism is known to protect against XSS

XSS: a side note



Attacker sends URL Link via email or embedded in another web page to the victim

Victim clicks link and requests page (with malicious script as parameters)

webserver returns the page embedded with the script the attacker chose

victims web browser then executes the script

jCSRF: performance

- GET requests
  - jCSRF only needs to generate a new token if the user does not have one already
- Same-origin POST requests
  - only needs to check if the authentication token is correct, which is a very low-cost operation
- Cross-origin POST requests
  - requires three additional GET requests: one to detect whether the target app is running jCSRF, one to fetch the iframe that requests the token, and one for the XmlHttpRequest that actually fetches the token
  - network delay is not negligible

Conclusion/Discussion

- jCSRF protects two things others do not
  - Dynamically created pages
  - Cross-origin requests

- Due to their use of javascript injection

- Small overhead except for cross-origin requests which incur a lot of network traffic
  - okay if list of authenticated domains is small and requests are sparse

IF JAVASCRIPT IS DISABLED, jCSRF is <u>FULLY INCOMPATIBLE</u>

- problem?