Cross-Site Request Forgeries: Exploitation and Prevention

William Zeller[⋆] and Edward W. Felten[⋆†]

[⋆]Department of Computer Science

[⋆]Center for Information Technology Policy

[†]Woodrow Wilson School of Public and International Affairs

Princeton University

{wzeller,felten}@cs.princeton.edu

Revision 10/15/2008: Noted that the New York Times has fixed the vulnerability described below. Also clarified that our server-side CSRF protection recommendations *do not* prevent the active network attacks described in [17].

The newest version of this paper can be found at `http://citp.princeton.edu/csrf/`

Abstract

Cross-Site Request Forgery (CSRF) attacks occur when a malicious web site causes a user's web browser to perform an unwanted action on a trusted site. These attacks have been called the "sleeping giant" of web-based vulnerabilities, because many sites on the Internet fail to protect against them and because they have been largely ignored by the web development and security communities. We present four serious CSRF vulnerabilities we have discovered on four major sites, including what we believe is the first published attack involving a financial institution. These vulnerabilities allow an attacker to transfer money out of user bank accounts, harvest user email addresses, violate user privacy and compromise user accounts. We recommend server-side changes (which we have implemented) that are able to completely protect a site from CSRF attacks. We also describe the features a server-side solution should have (the lack of which has caused CSRF protections to unnecessarily break typical web browsing behavior). Additionally, we have implemented a client-side browser plugin that can protect users from certain types of CSRF attacks even if a site has not taken steps to protect itself. We hope to raise the awareness of CSRF attacks while giving responsible web developers the tools to protect users from these attacks.

1 Introduction

Cross-Site Request Forgery[1] (CSRF) attacks occur when a malicious web site causes a user's web browser to perform an unwanted action on a trusted site. These attacks have been called the "sleeping giant" of web-based vulnerabilities [23], because many sites on the Internet fail to protect against them and because they have been largely ignored by the web development and security communities. CSRF attacks do not appear in the *Web Security Threat Classification* [12] and are rarely discussed in academic or technical literature.[2] CSRF attacks are simple to diagnose, simple to exploit and simple to fix. They exist because web developers are uneducated about the cause and seriousness of CSRF attacks. Web developers also may be under the mistaken impression that defenses against the better-known Cross-Site Scripting (XSS) problem also protect against CSRF attacks.

In Section 3, we present four serious CSRF vulnerabilities we have discovered on four major sites. These vulnerabilities allow an attacker to transfer money out of user bank accounts, harvest user email addresses, violate user privacy and compromise user accounts.

In Section 4.1, we recommend server-side changes (which we have implemented) that are able to completely protect a site from CSRF attacks. These recommendations have advantages over previously proposed solutions since they do not require server state and do not break typical web browsing behavior. Additionally, we have implemented a client-side browser plugin that can protect users from certain types of CSRF attacks (Section 4.2). The server-side protections allow a site to completely protect

[1]Cross-Site Request Forgery attacks are also known as *Cross-Site Reference Forgery*, *XSRF*, *Session Riding* and *Confused Deputy* attacks. We use the term CSRF because it appears to be the most commonly used term for this type of attack.

[2]A search for "cross site scripting" (which differs from CSRF) on the ACM Digital Library returned 72 papers, while a search for "xsrf OR csrf" returned only four papers. A search for "xss" on Safari Books Online (a collection of 4752 books on technology) showed the term appeared in 96 books, while "csrf OR xsrf" appeared in only 13 books.

itself from CSRF attacks, while the client-side protection allows users to take proactive steps to protect themselves from many types of CSRF attacks even if a site has not taken steps to protect itself. We hope to raise the awareness of CSRF attacks while giving responsible web developers the tools to protect users from these attacks.

2 Overview of CSRF

Figures 1, 2 and 3 show how CSRF attacks generally work.

Below we describe CSRF attacks in more detail using a specific example.

2.1 An Example

Let's consider a hypothetical example of a site vulnerable to a CSRF attack. This site is a web-based email site that allows users to send and receive email. The site uses implicit authentication (see Section 2.2) to authenticate its users. One page, `http://example.com/compose.htm`, contains an HTML form allowing a user to enter a recipient's email address, subject, and message as well as a button that says, "Send Email."

```
<form
action="http://example.com/send_email.htm"
method="GET">
Recipient's Email address:  <input
type="text" name="to">
Subject:  <input type="text" name="subject">
Message:  <textarea name="msg"></textarea>
<input type="submit" value="Send Email">
</form>
```

When a user of `example.com` clicks "Send Email", the data he entered will be sent to `http://example.com/send_email.htm` as a GET request. Since a GET request simply appends the form data to the URL, the user will be sent to the following URL (assuming he entered "bob@example.com" as the recipient, "hello" as the subject, and "What's the status of that proposal?" as the message):

```
http://example.com/send_email.htm?to=bob%
40example.com&subject=hello&msg=What%27s+the+
status+of+that+proposal%3F
```
[3]

The page `send_email.htm` would take the data it received and send an email to the recipient from the user. Note that `send_email.htm` simply takes data and performs an action with that data. It does not care where the request originated, only that the request was made. This means that if the user manually typed in the above URL into his browser, `example.com` would still send an email. For example, if the user typed the following three URLs into his browser, `send_email.htm` would send three emails (one each to Bob, Alice, and Carol):

```
http://example.com/send_email.htm?to=bob%
40example.com&subject=hi+Bob&msg=test
http://example.com/send_email.htm?to=alice%
40example.com&subject=hi+Alice&msg=test
http://example.com/send_email.htm?to=carol%
40example.com&subject=hi+Carol&msg=test
```

A CSRF attack is possible here because `send_email.htm` takes any data it receives and sends an email. It does not verify that the data originated from the form on `compose.htm`. Therefore, if an attacker can cause the user to send a request to `send_email.htm`, that page will cause `example.com` to send an email on behalf of the user containing any data of the attacker's choosing and the attacker will have successfully performed a CSRF attack.

To exploit this vulnerability, the attacker needs to force the user's browser to send a request to `send_email.htm` to perform some nefarious action. (We assume the user visits a site under the attacker's control and the target site does not defend against CSRF attacks.) Specifically, the attacker needs to *forge* a *cross-site request* from his site to `example.com`. Unfortunately, HTML provides many ways to make such requests. The `<img>` tag, for example, will cause the browser to load whatever URI is set as the *src* attribute, even if that URI is not an image (because the browser can only tell the URI is an image after loading it). The attacker can create a page with the following code:

```
<img src="http://example.com/send_email.htm?
to=mallory%40example.com&subject=Hi&msg=My+
email+address+has+been+stolen">
```

When the user visits that page, a request will be sent to `send_email.htm`, which will then send an email to Mallory from the user. This example is nearly identical to an actual vulnerability we discovered on the New York Times website, which we describe in Section 3.1.

CSRF attacks are successful when an attacker can cause a user's browser to perform an unwanted action on another site. For this action to be successful, the user must be capable of performing this action. CSRF attacks are typically as powerful as a user, meaning any action the user can perform can also be performed by an attacker using a CSRF attack. Consequently, the more power a site gives a user, the more serious are the possible CSRF attacks.

CSRF attacks can be successful against nearly every site that uses implicit authentication (see Section 2.2) and does not explicitly protect itself from CSRF attacks.

The *same-origin* policy (see Appendix B) was designed to prevent an attacker from accessing data on a third-party site. This policy does not prevent requests from being sent, it only prevents an attack from reading the data returned from the third-party server. Since CSRF attacks are the result of the requests sent, the *same-origin* policy does not protect against CSRF attacks.

[3]The URL data is encoded, turning @ into *%40*, *etc*

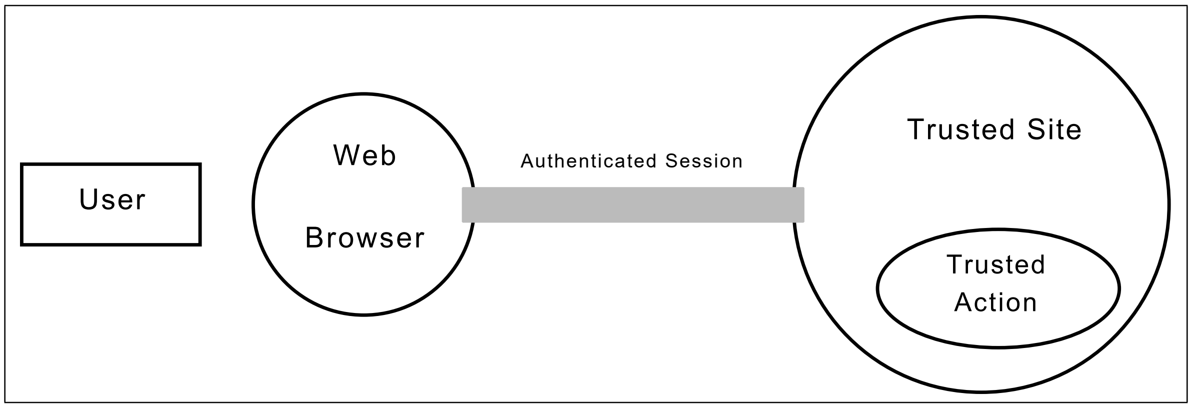

Figure 1: *Here, the Web Browser has established an authenticated session with the Trusted Site. Trusted Action should only be performed when the Web Browser makes the request over the authenicated session.*

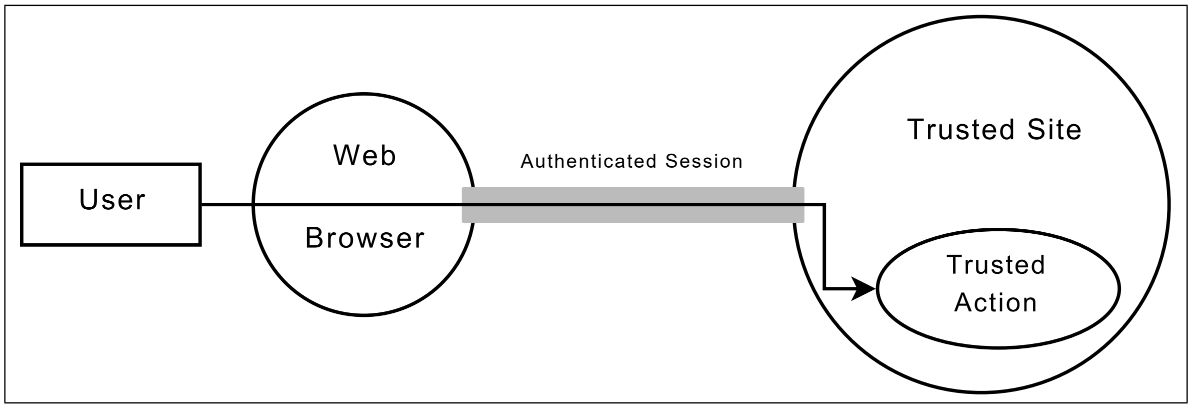

Figure 2: *A valid request. The Web Browser attempts to perform a Trusted Action. The Trusted Site confirms that the Web Browser is authenticated and allows the action to be performed.*

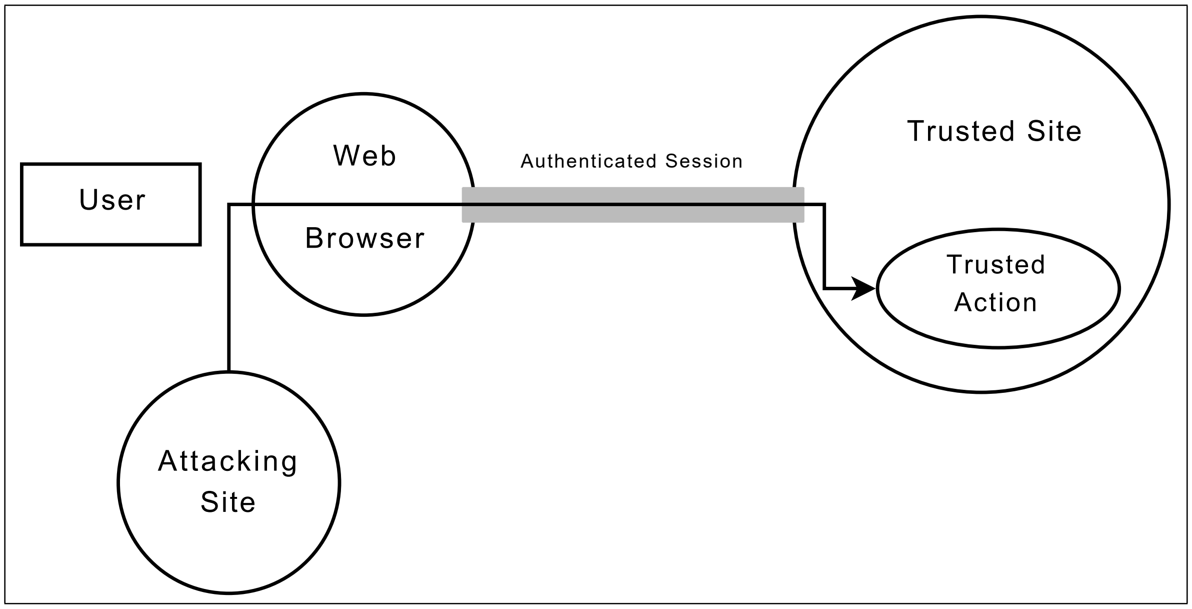

Figure 3: *A CSRF attack. The Attacking Site causes the browser to send a request to the Trusted Site. The Trusted Site sees a valid, authenticated request from the Web Browser and performs the Trusted Action. CSRF attacks are possible because web sites authenticate the web browser, not the user.*

2.2 Authentication and CSRF

CSRF attacks often exploit the authentication mechanisms of targeted sites. The root of the problem is that Web authentication normally assures a site that a request came from a certain user's browser; but it does not ensure that the user actually requested or authorized the request.

For example, suppose that Alice visits a target site T. T gives Alice's browser a cookie containing a pseudorandom session identifier sid, to track her session. Alice is asked to log in to the site, and upon entry of her valid username and password, the site records the fact that Alice is logged in to session sid. When Alice sends a request to T, her browser automatically sends the session cookie containing sid. T then uses its record to identify the session as coming from Alice.

Now suppose Alice visits a malicious site M. Content supplied by M contains Javascript code or an image tag that causes Alice's browser to send an HTTP request to T. Because the request is going to T, Alice's browser "helpfully" appends the session cookie sid to the request. On seeing the request, T infers from the cookie's presence that the request came from Alice, so T performs the requested operation on Alice's account. This is a successful CSRF attack.

Most of the other Web authentication mechanisms suffer from the same problem. For example, the HTTP BasicAuth mechanism [22] would have Alice tell her browser her username and password for T's site, and then the browser would "helpfully" attach the username and password to future requests sent to T. Alternatively, T might use client-side SSL certificates, but the same problem would result because the browser would "helpfully" use the certificate to carry out requests to T's site. Similarly, if T authenticates Alice by her IP address, CSRF attacks would be possible.

In general, whenever authentication happens implicitly—because of which site a request is being sent to and which browser it is coming from—there is a danger of CSRF attacks. In principle, this danger could be eliminated by requiring the user to take an explicit, unspoofable action (such as re-entering a username and password) for each request sent to a site, but in practice this would cause major usability problems. The most standard and widely used authentication mechanisms fail to prevent CSRF attacks, so a practical solution must be sought elsewhere.

2.3 CSRF Attack Vectors

For an attack to be successful, the user must be logged-in to the target site and must visit the attacker's site or a site over which the attacker has partial control.

If a server contains CSRF vulnerabilities and also accepts GET requests (as in the above example), CSRF attacks are possible *without the use of JavaScript* (for example, a simple `<img>` tag can be used). If the server only accepts POST requests, JavaScript is required to automatically send a POST request from the attacker's site to the target site.

2.4 CSRF vs. XSS

Recently much attention has been paid to *Cross-Site Scripting* (XSS) [20] vulnerabilities. A XSS attack occurs when an attacker injects malicious code (typically JavaScript) into a site for the purpose of targeting other users of the site. For example, a site might allow users to post comments. These comments are submitted by a user, stored in a database and displayed to all future users of the site. If an attacker is able to enter malicious JavaScript as part of a comment, the JavaScript would be embedded on any page containing the comment. When a user visits the site, the attacker's JavaScript would be executed with all the privileges of the target site. Malicious JavaScript embedded in a target site would be able to send and receive requests from any page on the site and access cookies set by that site. Protection from XSS attacks requires sites to carefully filter any user input to ensure that no malicious code is injected.

CSRF and XSS attacks differ in that XSS attacks require JavaScript, while CSRF attacks do not. XSS attacks require that sites accept malicious code, while with CSRF attacks malicious code is located on third-party sites. Filtering user input will prevent malicious code from running on a particular site, but it will not prevent malicious code from running on third-party sites. Since malicious code can run on third-party sites, protection from XSS attacks does not protect a site from CSRF attacks. If a site is vulnerable to XSS attacks, then it *is vulnerable* to CSRF attacks. If a site is completely protected from XSS attacks, it is most likely *still vulnerable* to CSRF attacks.

3 CSRF Vulnerabilities

In this section we describe four vulnerabilities we discovered. These attacks were found by surveying a list of about ten popular websites. Many of the sites we analyzed either had CSRF vulnerabilities or a history of vulnerabilities (*e.g.*, a web search would show a report of a CSRF vulnerability that has since been fixed). The fact that so many sites are vulnerable to CSRF attacks until a third-party discloses the problems shows that many site administrators are uneducated about the risks and existence of CSRF vulnerabilities.

We believe ING Direct, Metafilter and YouTube, and the New York Times have corrected the vulnerabilities we describe below. All four sites appear to have fixed the problem using methods similar to what we propose in Section 4.1.

3.1 The New York Times (nytimes.com)

The New York Times website is the "the #1 reaching newspaper site on the Web" [10].

We discovered a CSRF vulnerability in NYTimes.com that makes user email addresses available to an attacker. If you are a NYTimes.com member, abitrary sites can use this attack to determine your email address and use it to send spam or to identify you.

This attack takes advantage of NYTimes.com's "Email This" feature. "Email This" is a tool that allows a user to send a link to a NYTimes.com article by specifying a recipient's email address and optionally, a personal message. The recipient receives an email that looks something like this:

```
This page was sent to you by:  [USER'S EMAIL
ADDRESS]

Message from sender:
Thought you'd be interested in this.

NATIONAL DESK
Researchers Find Way to Steal Encrypted Data
By JOHN MARKOFF
A computer security research group has
developed a way to steal encrypted
information from computer hard disks.
```

To exploit this vulnerability, an attacker causes a logged-in user's browser to send a request to the NYTimes.com "Email This" page. The page accepting "Email This" requests does not protect against CSRF attacks, so the user's browser will cause a request to be sent to NYTimes.com that will trigger it to send an email to an address of the attacker's choosing. If the attacker changes the *recipient* email address to his own email address, he will receive an email from NYTimes.com containing the user's email address.

Exploiting this vulnerability is remarkably simple. Each article on NYTimes.com contains a link to the "Email This" page, which contains a form where the user enters a recipient's email address. This form also contains hidden variables which are unique for each article. Here is an example form:

```
<form
action="http://www.nytimes.com/mem/emailthis.html"
method="POST"
enctype="application/x-www-form-urlencoded">
<input type="checkbox" id="copytoself"
name="copytoself" value="Y">
<input id="recipients" name="recipients"
type="text" maxlength="1320" value="">
<input type="hidden" name="state" value="1">
<textarea id="message" name="personalnote"
maxlength="512"></textarea>
<input type="hidden" name="type" value="1">
<input type="hidden" name="url"
value="[...]">
<input type="hidden" name="title"
value="[...]">
<input type="hidden" name="description"
value="[...]">
...
</form>
```

Since NYTimes.com does not distinguish between GET and POST requests, the attacker can transform this form into a GET request that can later be used in an `<img>` tag. Transforming a form into a GET request involves appending each parameter to the URL's query string (in the form *NAME=VALUE*, separated by an ampersand).

Once the attacker has constructed the URL, he can set it to be the SRC attribute of an `<img>` tag. If a logged-in user of NYTimes.com visits any page containing this `<img>` tag, the browser will load the "Email This" page with the attacker's parameters, causing NYTimes.com to send an email to the attacker containing the user's email address. The attacker could store this email address for later abuse (*e.g.*, for spamming purposes) or use the email address to identify visitors to his own site. This could lead to serious privacy consequences, such as allowing operators of controversial sites (*e.g.*, political or illicit) to identify their users.

We verified this attack in Firefox 2.0.0.6, Opera 9.23 and Safari 3.0.3 (522.15.5). It does not work in Internet Explorer due to reasons described in Appendix A. We notified the New York Times *of this vulnerability in September 2007. This was fixed by Oct 1, 2008.*

3.2 ING Direct (ingdirect.com)

"ING DIRECT is the fourth largest savings bank in America with more than $62 billion in assets, delivering superior savings and mortgage services to over 4.1 million customers." [15]

We discovered CSRF vulnerabilities in ING's site that allowed an attacker to open additional accounts on behalf of a user and transfer funds from a user's account to the attacker's account. As we discuss in Section 2.2, ING's use of SSL does not prevent this attack. We believe this is the first published CSRF attack involving a financial institution.

Since ING did not explicitly protect against CSRF attacks, transferring funds from a user's accounts was as simple as mimicking the steps a user would take when transferring funds. These steps consist of the following actions:

1. The attacker creates a checking account on behalf of the user.[4]

[4]ING Direct allows checking accounts to be created instantly with any initial amount of money.

(a) The attacker causes the user's browser to visit ING's "Open New Account" page:

- A GET request to `https://secure.ingdirect.com/myaccount/INGDirect.html?command=gotoOpenOCA`

(b) The attacker causes the user's browser to choose a "single" account:

- A POST request to `https://secure.ingdirect.com/myaccount/INGDirect.html` with the parameters:

```
command=ocaOpenInitial&YES, I
WANT TO CONTINUE..x=44&YES, I
WANT TO CONTINUE..y=25
```

(c) The attacker chooses an arbitrary amount of money to initially transfer from the user's savings account to the new, fraudulent account:

- A POST request to `https://secure.ingdirect.com/myaccount/INGDirect.html` with the parameters:

```
command=ocaValidateFunding&PRIMARY
CARD=true&JOINTCARD=true&Account
Nickname=[ACCOUNT_NAME]&FROMACCT=
0&TAMT=[INITIAL_AMOUNT]&YES, I
WANT TO CONTINUE..x=44&YES, I
WANT TO
CONTINUE..y=25&XTYPE=4000USD
&XBCRCD=USD
```

...where **[ACCOUNT_NAME]** is the name of the account that the user will see and **[INITIAL_AMOUNT]** is the amount of money that will be transferred to the new account when it is opened. The account name can be any string and does not need to be known by the attacker before hand–it is simply a nickname that will be used for the new account.

(d) The attacker causes the user's browser to click the final "Open Account" button, causing ING to open a new checking account on behalf of the user:

- A POST request to `https://secure.ingdirect.com/myaccount/INGDirect.html` with the parameters:

```
command=ocaOpenAccount&Agree
ElectronicDisclosure=yes&AgreeTerms
Conditions=yes&YES, I WANT TO
CONTINUE..x=44&YES, I WANT TO
CONTINUE..y=25&YES, I WANT TO
CONTINUE.=Submit
```

2. The attacker adds himself as a payee to the user's account.

(a) The attacker causes the user's browser to visit ING's "Add Person" page:

- A GET request to `https://secure.ingdirect.com/myaccount/INGDirect.html?command=goToModifyPersonalPayee&Mode=Add&from=displayEmailMoney`

(b) The attacker causes the user's browser to enter the attacker's information:

- A POST request to `https://secure.ingdirect.com/myaccount/INGDirect.html` with the parameters:

```
command=validateModifyPersonalPayee
&from=displayEmailMoney&PayeeName
=[PAYEE_NAME]&PayeeNickname=&chk
Email=on&PayeeEmail=[PAYEE_EMAIL]
&PayeeIsEmailToOrange=true&Payee
OrangeAccount=[PAYEE_ACCOUNT_NUM]&
YES, I WANT TO CONTINUE..x=44
&YES, I WANT TO CONTINUE..y=25
```

...where **[PAYEE_NAME]** is the attacker's name, **[PAYEE_EMAIL]** is the attacker's email address and **[PAYEE_ACCOUNT_NUM]** is the attacker's ING account number.

(c) The attacker causes the user's browser to confirm that the attacker is a valid payee:

- A POST request to `https://secure.ingdirect.com/myaccount/INGDirect.html` with the parameters:

```
command=modifyPersonalPayee&from=
displayEmailMoney&YES, I WANT TO
CONTINUE..x=44 &YES, I WANT TO
CONTINUE..y=25
```

3. The attacker transfers funds from the user's account to his own account.

(a) The attacker causes the user's browser to enter an amount of money to send to the attacker:

- A POST request to `https://secure.ingdirect.com/myaccount/INGDirect.html` with the parameters:

```
command=validateEmailMoney&CNSPayID
=5000&Amount=[TRANSFER_AMOUNT]
&Comments=[TRANSFER_MESSAGE]&YES,
I WANT TO CONTINUE..x=44 &YES, I
WANT TO
CONTINUE..y=25&show=1&button=Send
Money
```

...where **[TRANSFER_AMOUNT]** is the amount of money to transfer from the user's account to the attacker's account and **[TRANSFER_MESSAGE]** is the message to include with the transaction.

(b) The attacker causes the user's browser to confirm that the money should be sent:

- A POST request to `https://secure.ingdirect.com/myaccount/INGDirect.html` with the parameters:

  ```
  command=emailMoney&Amount=
  [TRANSFER_AMOUNT]Comments=
  [TRANSFER_MESSAGE]&YES, I WANT
  TO CONTINUE..x=44&YES, I WANT TO
  CONTINUE..y=25
  ```

  …where **[TRANSFER_AMOUNT]** and **[TRANSFER_MESSAGE]** are the same values as 3 (a) above.

To exploit this attack, an attacker would create a page that made the above POST requests in succession using JavaScript. This would be invisible to the user.

This attack assumes the user has not added an additional payee to his ING Direct checking account. The attack could likely have been modified to work without this restriction.

We verified this attack in Firefox 2.0.0.3 and Internet Explorer 7.0.5. We did not test this attack in other browsers. We notified ING of this vulnerability and it has since been fixed.

3.3 MetaFilter (metafilter.com)

"MetaFilter is a weblog… that anyone can contribute a link or a comment to." It currently has over 50,000 users and over 3.5 million unique visitors each month [1].

We discovered a CSRF vulnerability in MetaFilter that allowed an attacker to take control of a user's account.

MetaFilter has a "Lost Password" page [6] which allows a user to request his password. Entering a username causes MetaFilter to send an email containing the user's current password to the email address associated with that user. This means that an attacker with the ability to change a user's email address could have used the "Lost Password" page to receive the user's password and used that password to take control of the user's account.

The CSRF attack we found allowed an attacker to change a user's email address. To exploit this attack, an attacker would cause the user's browser to send a request to the page that is used to update a user's profile. This page accepts the user's email address as an argument, which can be replaced with the attacker's address. An example attack would be the following HTML embedded on a page:

```
<img src="
http://metafilter.com/contribute/customize_
action.cfm?user_email=[ATTACKER'S_EMAIL]"/>
```

While this will change the email address of any logged-in user, the attacker will not know which user's account was modified. The attacker can discover this by taking advantage of another MetaFilter feature, which allows users to mark other users as "contacts." The attacker can use a CSRF similar to the above to cause a user to unknowingly add the attacker to his contact list.

We verified this attack in Firefox 2.0.0.6. It does not work in Internet Explorer due to reasons described in Appendix A. We did not test this attack in other browsers. We reported this vulnerability to MetaFilter and confirmed it was fixed within two days.

3.4 YouTube (youtube.com)

"YouTube is the leader in online video, and the premier destination to watch and share original videos worldwide through a Web experience" [13]. A June 2006 study found that "YouTube alone comprises approximately 20% of all HTTP traffic, or nearly 10% of all traffic on the Internet." [14]

We discovered CSRF vulnerabilities in nearly every action a user can perform on YouTube. An attacker could have added videos to a user's "Favorites," add himself to a user's "Friend" or "Family" list, sent arbitrary messages on the user's behalf, flagged videos as inappropriate, automatically shared a video with a user's contacts, subscribed a user to a "channel" (a set of videos published by one person or group) and added videos to a user's "QuickList" (a list of videos a user intends to watch at a later point). For example, to add a video to a user's "Favorites," an attacker simply needed to embed this `<img>` tag on any site:

```
<img src="http://youtube.com/watch_ajax?
action_add_favorite_playlist=1&video_
id=[VIDEO_ID]&playlist_id=&add_to_favorite=
1&show=1&button=AddvideoasFavorite"/>
```

An attacker could have used this vulnerability to impact the popularity of videos. For example, adding a video to a sufficient number of users' "Favorites" would have caused YouTube to show the video in its "Top Favorites" (a list of the videos "favorited" the greatest number of times). In addition to increasing a video's popularity, an attacker could have caused users to unknowingly flag a video as inappropriate in an attempt to cause YouTube to remove it from the site.

These attacks could also have been used to violate user privacy. YouTube allows users to make videos available only to friends or family. These attacks could have allowed an attacker to add himself automatically to a user's "Friend" or "Family" list, which would have given him access to any private videos uploaded by a user and restricted to these lists.

An attacker could have shared a video with a user's entire contact list ("Friends," "Family," *etc*). "Sharing" simply means sending a link to a video with an optional message attached. This message can include a link, meaning an attacker could force a user to include a link to a site containing the attack. Users receiving the message might click on this link, allowing the attack to spread virally.

We verified these attacks in Firefox 2.0.0.6. They do not work in Internet Explorer due to reasons described in Appendix A. We did not test these attacks in other browsers. We reported these vulnerabilities to YouTube and they appear to have been corrected.

4 Preventing CSRF

We have created two tools that can protect a large number of users from CSRF attacks. The first is a server-side tool which can completely protect a potential target site from CSRF attacks. The second is a client-side tool which can protect users from certain types of CSRF attacks. Table 1 describes exactly when users are protected by these different techniques. We also recommend features that should be part of a server-side solution. These recommendations have advantages over previously proposed solutions since they do not require server state and do not break typical web browsing behavior.

4.1 Server-Side Protection

Note: We assume below that an adversary cannot modify a user's cookies associated with a target site. The same-origin policy guarantees this to be the case unless the attacker is an active network attacker. The solution below does not protect against active network attackers (see [17] for more details).

Recently, many frameworks have been introduced that simplify web development in a variety of languages. Examples include Code Igniter [4] (PHP), Ruby on Rails [8] (Ruby), django [5] (Python), Catalyst [3] (Perl) and Struts [9] (Java). One major benefit of these frameworks is that CSRF protection can be built directly into the framework, protecting developers while freeing them from the need to implement protections on their own. CSRF protection implemented at the framework level would be subject to greater oversight and a lower chance of introducing bugs due to carelessness or a misunderstanding of CSRF.

Individual sites as well as frameworks can protect themselves from CSRF attacks by taking the following precautions:

1. **Allow GET requests to only *retrieve* data, not modify any data on the server**

   This change protects sites from CSRF attacks using `<img>` tags or other types of GET requests. Additionally, this recommendation follows RFC 2616 (HTTP/1.1):

   > In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe" [21].

   While this protection does not prevent CSRF on its own (since attackers can use POST requests), it can be combined with (2) to completely prevent CSRF vulnerabilities[5].

2. **Require all POST requests to include a pseudorandom value**

   When a user visits a site, the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine. The site should require every form submission to include this pseudorandom value as a form value and also as a cookie value. When a POST request is sent to the site, the request should only be considered valid if the form value and the cookie value are the same. When an attacker submits a form on behalf of a user, he can only modify the values of the form. An attacker cannot read any data sent from the server or modify cookie values, per the *same-origin* policy (see Appendix B). This means that while an attacker can send any value he wants with the form, he will be unable to modify or read the value stored in the cookie. Since the cookie value and the form value must be the same, the attacker will be unable to successfully submit a form unless he is able to guess the pseudorandom value.

3. **Use an pseudorandom value that is independent of a user's account**

   Pseudorandom values that are connected to a user's account fail to prevent the "Login CSRF" attack described in [17].

This form of server-side protection has the following characteristics:

- **Lightweight.** This solution requires no server-side state. The site's only responsibility is to generate a pseudorandom value (if none currently exists) and compare two values when a POST request is made, making this form of CSRF protection computationally inexpensive.

- **Parallel session compatible.** If a user has two different forms open on a site, CSRF protection should not prevent him from successfully submitting both forms. Consider what would happen if the site

[5]We assume an adversary cannot modify a user's cookies

	Unprotected User	User using our Firefox plugin
Target Server with no protections	Not Protected	Not Protected
Target Server that only accepts POST requests	Not Protected	Protected
Target Server that uses server-side protection	Protected	Protected

Table 1: User Protection.

This table shows when users are and are not protected from CSRF attacks. Our server-side recommendations protect every user of a site. Our client-side browser plugin protects a user when the server requires POST requests to be used.

were to generate a pseudorandom value each time a form was loaded, overwriting the old pseudorandom value. A user could only successfully submit the last form he opened, since all others would contain invalid pseudorandom values. Care must be taken to ensure CSRF protection does not break tabbed browsing or browsing a site using multiple browser windows. This solution prevents this problem by setting a site-wide cookie and using the same cookie for all forms for a certain amount of time.

- **Authentication agnostic.** This solution does not require that a specific type of authentication be used. It works with sites using cookie sessions, HTTP authentication, SSL authentication, or IP addresses to authenticate users.

The use of pseudorandom values in forms has been proposed before, but many of the proposed implementations do not have the above characteristics. For example, Johns and Winter [24] and Schreiber [27] require server state, while Shiflett [28] breaks tabbed browsing. To our knowledge, previously proposed solutions have not stressed the importance of working with typical browsing behavior.

Any framework that intercepts POST requests and "wraps" commands to generate `<form>` tags can build the above CSRF protection into the framework transparently. For example, if a framework requires a developer to call the function `form_open(...);` to generate a `<form ...>` tag, the framework can be modified to automatically generate a pseudorandom value each time a form is created:

```
<form ...>
<input type="hidden" name="csrf_value"
value="8dcb5e56904d9b7d4bbf333afdd154ca">
```

Additionally, the framework can handle setting associated cookie values and comparing the submitted value with the cookie value. If a framework adds this kind of CSRF protection, all users of the framework will be protected from CSRF attacks. Since this kind of CSRF protection is lightweight and agnostic to any authentication method the framework or developer might provide, we strongly advise that frameworks that intercept POST requests and provide functions to generate `<form>` tags implement this kind of CSRF protection and turn it on by default. The framework should provide developers with the ability to disable this protection if, e.g., they have implemented CSRF protection on their own or they do not want to require cookies.

We provide such a plugin for the Code Igniter framework. The plugin does not require the developer to modify any existing forms and will intercept (and validate the pseudorandom values of) POST requests as well as function calls that create `<form>` tags. The plugin also provides a function that allows CSRF tokens to be added to AJAX requests, although this requires developer intervention (Code Igniter does not have a standard way of performing AJAX requests). The plugin can be downloaded from our website[6].

4.2 Client-Side Protection

Since web browsers send the requests that allow an attacker to successfully perform CSRF attacks, client-side tools can be created to protect users from these attacks. One existing tool, *RequestRodeo* [24], works by acting as a proxy between the client and server. If *RequestRodeo* sees a request that it considers to be invalid, it strips authentication information from the request. While this works in many cases, it has some limitations. Specifically, it will not work when client-side SSL authentication is used, or when JavaScript is used to generate a part of the page (because *RequestRodeo* analyzes data as it passes through the proxy and before it is displayed by the browser).

We have developed a browser plugin that will protect users from certain types of CSRF attacks and overcomes the above limitations. We implemented our tool as an ex-

[6]Our Code Igniter Plugin: `http://www.cs.princeton.edu/~wzeller/csrf/ci/`

tension to the Firefox web browser. Users will need to download and install this extension for it to be effective against CSRF attacks.

Our extension works by intercepting every HTTP request and deciding whether it should be allowed. This decision is made using the following rules. First, any request that is not a POST request is allowed. Second, if the requesting site and target site fall under the *same-origin* policy (see Appendix B), the request is allowed. Third, if the requesting site is allowed to make a request to the target site using Adobe's *cross-domain policy* (see Appendix B), the request is allowed. If our extension rejects a request, the extension alerts the user that the request has been blocked using a familiar interface (the same one used by Firefox's popup blocker) and gives the user the option of adding the site to a whitelist.

Our extension only intercepts POST requests. This means our extension will not protect a user against a CSRF attack that works using GET requests. The only way to prevent this type of attack would be to either allow no *cross-domain* GET requests or to allow the user to only be logged-in to one site at a time, restrictions that users would likely find overly burdensome.

This Firefox extension is available for download on our website[7].

5 Related Work

The exposure CSRF attacks have received is largely due to the work of Chris Shiflett [28] of OmniTI and Jeremiah Grossman [23] of WhiteHat Security. Burns [19] and Schreiber [27] provide comprehensive introductions to CSRF attacks, but do not describe working vulnerabilities. Johns and Winter [24] describe *RequestRodeo*, a client-side protection against CSRF attacks using an HTTP proxy. This approach has some limitations and they describe a browser plugin similar to ours as possible future work. They have expanded on this work in [25] by implementing a *restricted local network* which prevents CSRF attacks against local resources.

Server-side protections exist that are similar to our recommendations, but the lack of standard requirements has caused unnecessary problems. As mentioned, Johns and Winter [24] and Schreiber [27] require server state, while Shiflett [28] breaks tabbed browsing. Jovanovic et al. [26] have created a method to retrofit legacy applications with CSRF protections by adding a proxy between the web server and the web application. These protections require all data be buffered and links in the application modified. They also require certain application calls to be rewritten. This solution is effective when the native application cannot be rewritten, but not as effective as adding CSRF protection to the application directly. This solution is aimed at administrators who want to protect applications on their servers from CSRF attacks, while our solution is aimed at web application and framework developers who want to add CSRF protection directly to their programs.

[7]Our CSRF Firefox Plugin: `http://www.cs.princeton.edu/~wzeller/csrf/protector/`

6 Future Work

Given the prevalence of CSRF vulnerabilities, an automated method to scan for these problems would be very useful. Bortz and Boneh [18] describe *Cross-Site Timing Attacks* and suggest that they may be combined with CSRF attacks to further compromise user privacy.

Our client-side browser plugin is a Firefox extension that only works with Firefox. Similar plugins could be written for other browsers. Likewise, the server-side methods used by our CodeIgniter extension could be easily implemented in other frameworks.

Our client-side browser plugin appears to be the first implementation of Adobe's *cross-domain* policy outside of Adobe's own Flash program. More work could be done to see if carefully adopting this policy in other contexts would increase user protection and site flexibility.

7 Conclusion

CSRF attacks are relatively simple to diagnose, exploit and fix. Sites can be analyzed in a matter of seconds; attacks can be constructed in a matter of minutes. The most plausible explanation for the prevalence of these attacks is that web developers are unaware of the problem or think (mistakenly) that defenses against the better-known cross-site scripting attacks also protect against CSRF attacks. We hope the attacks we have presented show the danger of CSRF attacks and help web developers to give these attacks the attention they deserve. Once web developers are made aware of CSRF attacks, they can use tools like the ones we have created to protect themselves.

We suggest the creators of frameworks add CSRF protection to their frameworks, thereby protecting any site built on top of such a framework. Adding CSRF protection at the framework level frees developers from duplicating code and even the need to understand CSRF attacks in detail (although understanding these attacks is recommended). Until every site is protected from CSRF attacks, users can take steps to protect themselves using our browser plugin for Firefox. Similar plugins could be written for other browsers.

The root cause of CSRF and similar vulnerabilities probably lies in the complexity of today's Web protocols, and the gradual evolution of the Web from a data presentation facility to a platform for interactive services. As more capabilities are added to browser clients, and as more sites involve sophisticated programming and client-server interactive services, CSRF and related attacks will become more prevalent unless defenses are adopted. As the complexity of Web technologies continue to increase, we can expect further new categories of attacks to emerge.

References

[1] About Metafilter. `http://www.metafilter.com/about.mefi`.

[2] Allowing cross-domain data loading. `http://livedocs.adobe.com/flash/9.0/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00001085.html`.

[3] Catalyst. `http://www.catalystframework.org/`.

[4] Code Igniter. `http://www.codeigniter.com/`.

[5] django. `http://www.djangoproject.com/`.

[6] Metafilter: Lost Password? `http://www.metafilter.com/login/lostpassword.mefi`.

[7] Privacy in Internet Explorer 6. `http://msdn2.microsoft.com/en-us/library/ms537343.aspx`.

[8] Ruby on Rails. `http://www.rubyonrails.org`.

[9] Struts. `http://struts.apache.org/`.

[10] The New York Times: Media Kit 2007. `http://www.nytimes.whsites.net/mediakit/pages/d_aud_target.html`.

[11] The Same-Origin Policy. `http://livedocs.adobe.com/flash/9.0/main/wwhelp/wwhimpl/common/html/wwhelp.htm`.

[12] Web Security Threat Classification. `http://www.webappsec.org/projects/threat/`.

[13] YouTube Fact Sheet. `http://www.youtube.com/t/fact_sheet`.

[14] Ellacoya Data Shows Web Traffic Overtakes Peer-to-Peer (P2P) as Largest Percentage of Bandwidth on the Network. `http://www.ellacoya.com/news/pdf/2007/NXTcommEllacoyaMediaAlert.pdf`, Jun 2006.

[15] ING Press Release. `http://www.rsa.com/press_release.aspx?id=7220`, Aug 2006.

[16] Alexa Top Sites. `http://www.alexa.com/site/sales`, Sep 2007.

[17] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS*, 2008.

[18] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 621–628, New York, NY, USA, 2007. ACM Press.

[19] J. Burns. Cross Site Reference Forgery: An introduction to a common web application weakness. `http://www.isecpartners.com/documents/XSRF_Paper.pdf`, 2005.

[20] D. Endler. The Evolution of Cross Site Scripting Attacks. `http://cgisecurity.com/lib/XSS.pdf`, May 2002.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999.

[22] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication, 1999.

[23] J. Grossman. CSRF, the sleeping giant. `http://jeremiahgrossman.blogspot.com/2006/09/csrf-sleeping-giant.html`, Sep 2006.

[24] M. Johns and J. Winter. RequestRodeo: Client Side Protection against Session Riding. In F. Piessens, editor, *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5 – 17. Departement Computerwetenschappen, Katholieke Universiteit Leuven, May 2006.

[25] M. Johns and J. Winter. Protecting the Intranet Against "JavaScript Malware" and Related Attacks. In *DIMVA*, 2007.

[26] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross Site Request Forgery Attacks. *Securecomm and Workshops, 2006*, pages 1–10, Aug. 28 2006-Sept. 1 2006.

[27] T. Schreiber. Session Riding: A Widespread Vulnerability in Today's Web Applications. `http://www.securenet.de/papers/Session_Riding.pdf`, 2004.

[28] C. Shiflett. Security Corner: Cross-Site Request Forgeries. `http://shiflett.org/articles/cross-site-request-forgeries`, Dec 2004.

[29] C. Shiflett. The crossdomain.xml Witch Hunt. `http://shiflett.org/blog/2006/oct/the-crossdomain.xml-witch-hunt`, Oct 2006.

[30] C. Shiflett. The Dangers of Cross-Domain Ajax with Flash. `http://shiflett.org/blog/2006/sep/the-dangers-of-cross-domain-ajax-with-flash`, Sep 2006.

A Internet Explorer and CSRF

Cookies can be used to track users across multiple websites. For example, suppose an advertiser hosts an image (an ad) on his server that is included by a large number of publisher sites. The advertiser could set a cookie when the image is displayed, which would allow the advertiser to identify a single user as he visits different publisher sites. That is, when the user vists a publisher site and loads the advertiser's image, his cookie would be sent back to the advertiser and uniquely identified. Advertisers can use these cookies to compile data about users' surfing habits.

Concern over such an adverse impact of cookies on user privacy led to the creation of the *Platform for Privacy Preferences (P3P)*. P3P "provides a common syntax and transport mechanism that enables Web sites to communicate their privacy practices to Internet Explorer 6 (or any other user agent)" [7]. Beginning with Internet Explorer 6, Microsoft began requiring that all sites include a P3P policy in order to receive third-party cookies.

According to Microsoft:

> Advanced cookie filtering works by evaluating a Web site's privacy practices and deciding which cookies are acceptable based on the site's compact policies and on the user's own preferences. In the default settings, cookies that are used to collect personally identifiable information and do not allow users a choice in their use are considered "unsatisfactory." By default, unsatisfactory cookies are deleted in the first-party context when the browsing session ends and are **rejected in the third-party context** [7].

(Note that P3P policies are not verified. If a site *claims* to have an acceptable policy, Internet Explorer allows third-party cookies.)

Suppose a user is on a page that contains an image located on a third-party site. In the context of P3P, the third-party site is potentially dangerous while the page the user is on is considered safe. With CSRF vulnerabilities, the opposite is true—the page the user is on is potentially dangerous while the third-party site is considered safe (and a potential target of attack). When Internet Explorer considers a third-party site to be dangerous, it prevents cookes from being sent to that site. This effectively prevents CSRF attacks when "session cookies" are used, because Internet Explorer is stripping authentication information from cross-site requests.

Internet Explorer's P3P policy has an interesting effect on CSRF vulnerabilities. Sites *with* valid P3P policies are *not* protected against CSRF attacks (Internet Explorer considers these sites safe and allows cookies) while sites *without* policies *are* protected (Internet Explorer considers these sites unsafe and blocks cookies) from CSRF attacks. Note that this only applies to CSRF vulnerabilities affecting sites using cookies for authentication. Sites using other types of authentication may still be vulnerable to CSRF attacks.

To summarize, Internet Explorer's use of P3P results in users of IE being protected from CSRF attacks when "session cookie" authentication is used and when target sites do not implement P3P policies. This "protection" is an unintended consequence of P3P policies and should not be exclusively used to prevent CSRF attacks. Sites should instead implement our server-side recommendations, as described in Section 4.1.

B The Same-Origin Policy

Web browsers have the difficult task of allowing users to maintain secure, private connections with multiple websites while also allowing access to untrusted sites that contain untrusted code. Additionally, sites are able to load resources from different domains. For example, site `a.com` can load images or JavaScript from `b.com` using `<img>` or `<script>` tags respectively. However, if a user is logged-in to a trusted site, an untrusted third-party should obviously not be able to read the contents of the trusted site. The desire to allow untrusted sites to display data from an external site while still maintaining the privacy of this data led to the creation of the *same-origin* policy [11]. This policy defines both the meaning of "origin" and the site's capabilities when accessing data from a different origin. The policy considers "two pages to have the same origin if the protocol, port (if given), and host are the same for both pages" [11]. According to the same-origin policy, a site cannot read or modify a resource from a different origin. It can, however, send a request for a resource from a different origin. Therefore, while `evil.com` can include the image `http://trusted.com/image.gif` in its site using the `<img>` tag, it cannot read the pixel data of this image. Similarly, while `evil.com` can include `http://trusted.com/private.htm` in its site using the `<iframe>` tag, it cannot access or modify the contents of the page displayed by the browser.

The same-origin policy only prevents a third-party site from *reading* data from another site, it does not prevent these third-party sites from *sending requests*. Since CSRF attacks are caused by requests being sent (causing some action to be performed on the server-side), the same-origin policy does not prevent CSRF attacks. Instead, it only protects the privacy of the data on third-party sites.

Sites sometimes find it useful or necessary to communicate across different domains. Adobe has proposed a mechanism, called the *cross-domain policy* [2], that would allow its Flash plugin to communicate (send and receive data) with different domains in certain cases. This mechanism is currently only used by Flash. Specifically, a site can specify which third-party sites can access it. A third-party site can only contact a trusted site if that trusted site lists the third-party site in its cross-domain policy file. The following example cross-domain policy file allows access to requests originating from www.friendlysite.com, *.trusted.com, and the IP address 64.233.167.99. These files are named *crossdomain.xml* and placed at the root of the domain.

```
<?xml version="1.0"?>
<cross-domain-policy>
<allow-access-from
domain="www.friendlysite.com" />
<allow-access-from domain="*.trusted.com" />
<allow-access-from domain="64.233.167.99" />
</cross-domain-policy>
```

Suppose the above file were located at `http://trusted.com/crossdomain.xml`. If a request were made by `evil.com` to `http://trusted.com/private.htm` using Flash, Flash would first load `http://trusted.com/crossdomain.xml` to verify that `evil.com` is listed as a trusted domain. Since it is not in the list, the request would be blocked. On the other hand, Flash would allow the same request from `www.friendlysite.com`, since it exists in the list of allowed domains.

When used properly, Adobe's cross-domain policy allows both more protection against CSRF attacks than the same-origin policy (the request cannot even be initiated unless a matching `crossdomain.xml` is found) and more flexibility (cross-domain communication is allowed if the target site trusts the initiating site). However, the cross-domain policy is often used improperly, where a target site puts an "accept all" clause. This allows third-party access from any site, whether malignant or benign. This improper and extremely dangerous use of `crossdomain.xml` files is even perpetuated by what appears to be an Adobe affiliated[8] site, `crossdomainxml.org`. This site provides an example of this "accept all" cross-domain policy file, with absolutely no explanation of the dangers involved in using this policy file. For more information on the dangers of using this type of cross-domain policy file, see Chris Shiflett ([30] and [29]).

We analyzed 500 top websites [16] and found 143 using `crossdomain.xml` policy files. Of those 143 sites, 47 sites accept all connections from third-party sites, possibly resulting in CSRF vulnerabilities.

Adobe's cross-domain policy can be effective and safe, when used with care. However, care must be taken to explain the dangers of the "accept all" solution.

[8]The domain `crossdomainxml.org` is registered to Theodore E Patrick of PowerSDK Software Corp who claims to be a "Technical Evangelist for Flex at Adobe Systems" on his LinkedIn profile (`http://www.linkedin.com/in/tedpatrick`)