SOMA: Mutual Approval for Included Content in Web Pages

Terri Oda Glenn Wurster Paul Van Oorschot Anil Somayaji
School of Computer Science, Carleton University
{toda,gwurster,paulv,soma}@scs.carleton.ca

SCS Technical Report TR-08-07

April 15, 2008

Abstract

Unrestricted information flows are a key security weakness of current web design. Cross-site scripting, cross-site request forgery, and other attacks typically require that information be sent or retrieved from arbitrary, often malicious, web servers. In this paper we propose *Same Origin Mutual Approval* (SOMA), a new policy for controlling information flows that prevents common web vulnerabilities. By requiring site operators to specify approved external domains for sending or receiving information, and by requiring those external domains to also approve interactions, we prevent page content from being retrieved from malicious servers and sensitive information from being communicated to an attacker. SOMA is compatible with current web applications and is incrementally deployable, providing immediate benefits for clients and servers that implement it. SOMA has an overhead of one additional HTTP request per domain accessed and can be implemented with minimal effort by application and web browser developers. To evaluate our proposal, we have developed a Firefox SOMA add-on, licensed under the GNU GPL.

Keywords: Web security, JavaScript, same origin policy, cross-site scripting, cross-site request forgery, cross-domain information flow

1 Introduction

Current web pages are more than collections of static information: they are a synthesis of code and data often provided by multiple sources that are assembled and run in the browser. Users generally trust the web sites they visit; however, external content may be untrusted, untrustworthy, or even malicious. Such malicious inclusions can initiate drive-by downloads [23], misuse a user's credentials [13], or even initiate distributed denial-of-service attacks [20].

One common thread in these scenarios is that the browser must communicate with web servers that normally wouldn't be contacted. Those servers may be controlled by an attacker, may be victims, or may be unwitting participants; whatever the case, information should not be flowing between the user's web browser and these sites.

In this paper, we propose a policy for constraining communications and inclusions in web pages. This policy, which we call *Same Origin Mutual Approval* (SOMA), requires the browser to check that both the owner of the page and the third party content provider approve of the inclusion before any communication is allowed (including adding anything to a page). This "tightening" of same origin policy prevents attackers

from loading malicious content from arbitrary web sites and restricts their ability to communicate sensitive information. While attacks such as cross-site scripting are still possible, with SOMA they must be mounted from domains trusted by the originating domain. Because attackers have much less control over this small subset than they do over other arbitrary hosts on the Internet, SOMA can prevent the exploitation of a wide range of vulnerabilities in web applications.

In addition to being effective, SOMA is also a practical proposal. To participate in SOMA, browsers have to make minimal code changes and web sites must create small, simple policy files. Sites and browsers participating in SOMA can see benefits immediately, while non-participating sites and browsers continue to function as normal. These characteristics facilitate incremental deployment, something that is essential for any change to Internet infrastructure.

We have implemented SOMA as an extension for Mozilla Firefox 2, one that can be run in any regular installation of the Firefox browser. In testing with this browser and simulated SOMA policy files for over 500 main pages on different sites, we have found no compatibility issues with current web sites. The policy files for these sites have been, with only a few exceptions, extremely easy to create and cause no compatibility issues. Simulated attacks are also appropriately blocked. To retrieve policy files, SOMA requires an extra web request per new domain visited. As we explain in Section 5, such overhead is minimal in practice. For these reasons, we argue that SOMA is a practical, easy to adopt, and effective proposal for improving the security of the modern web.

The remainder of this paper proceeds as follows. Section 2 gives background on current web security rules and attacks on modern web pages. Section 3 details the proposed Same Origin Mutual Approval design, which we then evaluate in Section 4. Our prototype and the results of testing in the browser are described in Section 5. We discuss some alternative JavaScript security proposals and other related work in Section 6. Section 7 concludes.

2 Background and Motivation

Web browsers are programs that regularly engage in extensive cross-domain communication. In the course of a user viewing a web page, they will retrieve images from one server, advertisements from another, and post a user's responses to a third. In this way the browser serves as a dynamic, cross-domain communications nexus. While such promiscuity may be permissible when combining static data, to maintain security, restrictions must be placed upon executable content.

JavaScript has two main security features that limit the potential damage of malicious scripts; the sandbox and the same origin policy. The sandbox prevents JavaScript code from affecting the underlying system (assuming there are no bugs in the implementation) or other web browser instances (including other tabs). Each page is contained within its own sandbox instance. The same origin policy [28] helps to define what can be manipulated within this sandbox and how sandboxed code can communicate with the outside world. The same origin policy is designed to prevent documents or scripts loaded from one "origin" from getting, or setting properties of, content loaded from a different origin (with a special case involving subdomains). The origin is defined as the protocol, port, and host from which the content originated. While scripts from different origins are not allowed to access each other's source, the functions in one script can be called from another script in the same page even if the two scripts are from different domains. JavaScript code has different access restrictions depending on the type of content being loaded. For example, it can fetch (make a request for) HTML, but it can only read and modify the information it gets as a result if the HTML falls within the same origin. These restrictions are summarized in Table 1.

Any script included onto a page inherits the origin of that page. This means that if a page from `http:`

Content Type	Permissions			
	Fetch	**Read**	**Modify**	**Execute**
Images	YES	SO	SO	NO
HTML	YES	SO	SO	NO
JavaScript	YES	SO	YES	YES
Styles	YES	SO	YES	NO
Audio/Video	YES	Plugin Dependant		NO

Table 1: Current restrictions on JavaScript access to other content (permissions denoted SO are dictated by the Same Origin Policy)

`//example.com` includes a script from `advertiser.com`, this script is considered to have the origin `http://example.com`. This allows the script to modify the web page from `example.com`. It is important to note that many scripts, including scripts dealing with embedding advertisements, require this ability. The script cannot subsequently read or manipulate content originating from `advertiser.com` directly; it can only read and manipulate content from `example.com`, or content which has inherited that origin.

While the sandbox and same origin policy protect the host and prevent many types of network communication, opportunities for recursive script inclusion, unrestricted outbound communication, cross-site request forgery, and cross-site scripting allow considerable scope for security vulnerabilities. We explore each of these issues below.

2.1 Recursive script inclusion

The same origin policy states that scripts can read or modify any part of a page with a matching origin. This includes allowing scripts to add additional script tags to the document. These new scripts are then loaded into the page, and also gain read and modify access to any content coming from the origin.

A page creator could choose to include content only from sources they deem trustworthy, but this does not mean that all content included will be directly from those sources. Any script loaded from a "trustworthy" domain can subsequently load content from *any* domain. Unfortunately, trust is not transitive, even if JavaScript treats it that way. Besides the risk of an intentionally malicious script loading additional, dangerous code, there is also the accidental risk of a "trustworthy" domain inadvertently loading malicious content. Even well-known, legitimate advertising services have been tricked into distributing malicious code [29, 25].

2.2 Unrestricted outbound communication

While the same origin policy restricts how content from another domain can be used, it does not stop any content from other domains from being requested and loaded into the origin site. These requests for content can be abused to send information out to any arbitrary domain.

One common attack involves cookie-stealing. A script reads cookie information from the user's browser and uses it as part of the URL of a request. This request could be for something innocuous, such as an extra image, as shown in Figure 1.

Such cookie information could then be used by the remote server to gain access to the user's session, or to get other information about the user. Any information that can be read from the document could be sent out in a similar manner, including credit card information, personal emails, or username and password

```
var image = new Image();
image.src= 'http://attacker.com/log.php?cookie='
  + encodeURIComponent(document.cookie);
```

Figure 1: Simple cookie-stealing JavaScript code which sends data to attacker.com

pairs. Even if a user does not hit "submit" on a form, any information they enter can be read by JavaScript and potentially retransmitted.

2.3 Cross-site request forgery (XSRF or CSRF)

The information theft techniques described in the previous section can be used to launch a *cross-site request forgery* (XSRF or CSRF) attack [5]. Some URLs, when requested, cause an action to be performed on the web server: a post is made, a friend is added, a vote is cast. Providing easy links for these actions is very useful for the web developer who may want to include them in a menu or elsewhere on the page. What happens, however, if one of these links is used as the URL for an image? Even though nothing has been clicked, that action will still be performed on behalf of the logged-in user because the URL is requested when the browser attempts to get the supposed image. Cross-site request forgery occurs when the user visits a web page which accesses a URL that performs an action (using that user's privileges) on another web page (even if the user never sees the URL being loaded).

2.4 Cross-site scripting (XSS)

While no precise definition of cross-site scripting seems to be universally accepted, the core concept behind cross-site scripting (XSS) is that of a security exploit in which an attacker inserts code onto a page returned by an unsuspecting web server [1, 2]. This code may be stored or reflected, it may contain JavaScript or just HTML, and it may use third party sites as sources or rely only upon the resources of the targeted server. With such ambiguity, it is possible to have a cross-site scripting attack which neither uses scripting nor is cross-site. Typically, however, XSS attacks involve JavaScript code engaging in cross-domain communication with malicious web servers.

Code injection for cross-site scripting usually occurs because user-inputted data is not sufficiently sanitized before being stored and/or displayed to other users. Although the existence of such vulnerabilities is not a flaw in the same origin policy, per se, the same origin policy does allow the injected code access to content of the originating site. Specifically, it can then steal information associated with that domain or perform actions on behalf of the user.

Some existing proposals to address cross-site scripting and other JavaScript security issues are described in greater detail in Sections 6. Here we note that no current proposal targets the cross-domain communication involved in most JavaScript exploits.

3 SOMA Design

The Same Origin Mutual Approval (SOMA) policy aims to tighten the same origin policy so that it can better handle exploits as discussed in Section 2, including cross-site scripting and cross-site request forgery. SOMA requires that both the origin web site and the site providing included content approve of the request before the browser allows any external content to be fetched for a page. Adding these extra checks gives site

Content Type	Permissions			
	Fetch	**Read**	**Modify**	**Execute**
Images	SOMA	SO	SO	NO
HTML	SOMA	SO	SO	NO
JavaScript	SOMA	SO	YES	YES
Styles	SOMA	SO	YES	NO
Audio/Video	SOMA	Plugin Dependant		NO

Table 2: Restrictions on JavaScript access to other content with SOMA (permissions denoted SO are dictated by the long-standing same origin policy)

operators much more control over what gets included into or from their sites. These changes are shown in Table 2. While the differences (relative to Table 1) are all in the Fetch column, a "fetch" can also be used to leak (send out) information such as cookies, as discussed earlier.

A key idea behind SOMA is that security policy should be decided by site operators, who have a vested interest in doing it correctly and the knowledge necessary to create secure policies, rather than end users. Having said that, we cannot expect site operators to create complex policies—their time and resources are limited. Thus SOMA works at a level of granularity that is both easy to understand and specify, that of DNS domains and URLs.

3.1 Threat Model

We assume that site administrators have the ability to create and control top-level URLs (static files or scripts) and that web browsers will follow the instructions specified at these locations precisely. In contrast, we do assume that the attacker controls arbitrary web servers and some of the content on legitimate servers (but not their policy files or their server software). Our goal is to prevent a web browser from communicating with a malicious web server when a legitimate web site is accessed, even if the content on that site or its partners has been compromised.

These assumptions mean that we do not address situations where an attacker compromises a web server to change policy files, compromises a web browser to circumvent policy checks, or performs intruder-in-the-middle attacks to intercept and modify communications. Further, we do not address the problem of users visiting malicious web sites directly, say as part of a phishing attack. While these are all important types of attacks, by focusing on the problem of unapproved communication we can create a simple, practical solution that addresses the security concerns we described in Section 2. Mechanisms to address these other threats largely complement rather than overlap with the protections of SOMA (see Section 6).

3.2 Manifest files

The first part of SOMA we discuss is the manifest file, which contains a list of domains which the origin domain wishes to allow included content. This idea is similar to the manifests provided in Tahoma [7]. This manifest file is always stored in the root directory and will have the name `soma-manifest`.

For example, the manifest file for `maps.google.com` would be found at `http://maps.google.com/soma-manifest` and might appear similar to Figure 2. If this file was set, the browser will enforce that only content from those locations could be embedded in a page coming from `maps.google.com`. Note that each location definition includes protocol, domain and optionally port (the default one for the

protocol is used if none is specified), so that origins are defined the same way as they are in the current same origin policy.

```
http://maps.l.google.com
http://www.google.com
http://mt0.google.com
http://mt1.google.com
http://mt2.google.com
http://mt3.google.com
```

Figure 2: Sample manifest for `maps.google.com`

If the origin A has a manifest that contains B, we denote this using $A \Subset B$. This symbol is chosen as a visual way to indicate that A is the origin (the outer cup) and B is a content provider web site for that origin (the inner circle). Similarly, if A's manifest does not include C, we denote that as $A \not\Subset C$. If $A \not\Subset C$ then the then browser will not allow anything in the pages from A to contact the domain C, thus code, images, iframes, or any other content will not be loaded from C.

By convention, it is not necessary to include the origin domain itself in the manifest file as inclusions from the origin are assumed to be allowed.

3.3 Approval files

The approval files provide the other side of the mutual approval by allowing domains to indicate sites which are allowed to include content from them. SOMA approval files are similar in function to Adobe Flash's `crossdomain.xml` [4] but differs in that it is not a single static file containing information about all approved domains. Instead, it is a script that provides a `YES`/`NO` response given a domain as input.

We use a script to prevent easy disclosure of the list of approved domains, since such a list could be used by an attacker (e.g. to determine which sites could be used in a XSRF attack or to determine business relationships). Attackers could still generate such a list by constantly querying `soma-approval`, but if they knew a list of domains to guess, they could just as easily visit those domains and see if they included any content from the target content provider. In addition, the smaller size of the approval responses containing simple `YES`/`NO` answers may provide a modest performance increase on the client side relative to the cost of loading a complete list of approved sites (especially for highly connected sites such as ad servers).

To indicate that A.com is allowed to load content from B.org, B.org needs to provide a script with the filename `/soma-approval` which returns `YES` when invoked through `http://B.org/soma-approval?d=A.com`. Negative responses can be indicated in a similar manner with the text of `NO`. If a negative response is received, then the browser refuses to load any content from B.org into a page from A.com. If no file with the name `soma-approval` exists, then we assume a default permissive behavior, described in greater detail in Section 3.6.

To reject all approval requests, `soma-approval` need only be a static file containing the string `NO`. Similarly, a static `soma-approval` with the word `NO` suffices to approve all requests.

An alternative proposal that avoids the need for a script involves allowing `soma-approval` to be a directory containing files for the allowed domains. Unfortunately, in order to handle our default permissive behavior, we would now require two requests: one to see if the soma-approval directory exists and another to see if the domain-specific file exists. Since most of the overhead of SOMA lies in the network requests (as shown in Section 5), we believe the better choice is to require a script.

```
<?php
   $site_policy = array(
      'A.com' => 'YES',
      'C.net' => 'YES');

   if (isset($site_policy[$_GET['d']])) {
      print $site_policy[$_GET['d']];
   } else {
      print 'NO';
   }
?>
```

Figure 3: Simple `soma-approval` script written in PHP

A sample `soma-approval` script, written in PHP, is shown in Figure 3. This script uses an array to store policy information at the top of the file then outputs the policy as requested, defaulting to `NO` if no policy has been defined. In this example, A.com and C.net are the only approved domains.

The symbols used for denoting approval are similar to those used for denoting inclusion in the manifest. If B approves of content from its site being included into a page with origin A we show this using $B \circledcirc\!\!\!\!\rightarrow A$. Again, since B is the content provider it is connected to the small inner circle, and the origin A is connected to the outer cup. If B does not approve of another domain C, this is denoted using $B \not\circledcirc\!\!\!\!\rightarrow C$. If $B \not\circledcirc\!\!\!\!\rightarrow C$ then the browser will refuse to allow the page from C to contact B in any way. No scripts, images, iframes or other content from B will be loaded for the web page at C.

It is important to note that $B \circledcirc\!\!\!\!\rightarrow A$ is not the same as, nor does it necessarily imply, that $A \circledcirc\!\!\!\!- B$. It is possible for one party to allow the inclusion and the other to refuse. Content is only loaded if both parties agree (i.e. $(A \circledcirc\!\!\!\!- C) \wedge (C \circledcirc\!\!\!\!\rightarrow A)$).

3.4 Content inclusions

Figure 4 illustrates inclusions currently allowed within the same origin policy. The web page itself indicates the content it needs, be it images, text or JavaScript code. The web browser retrieves this content and builds the page using it. It is important to note that it is the web page (running in the web browser) and not the web server that indicates the content, as scripts that are executed on the page may request additional content.

The additional constraints added by SOMA are illustrated in Figure 5. Rather than allowing all inclusions as requested by the web page, the modified browser checks first to see if both the page's web server and the external content's web server approve of each other. In Figure 5, web server A is the source of the web page to be displayed. A has a manifest that indicates that it approves of including content from both B and C ($A \circledcirc\!\!\!\!- B$ and $A \circledcirc\!\!\!\!- C$). When the browser is asked to include content from B in the page from A, it makes a request to B to determine if $B \circledcirc\!\!\!\!\rightarrow A$ (B approves of A incorporating its content). In the example, B approves and its content is included on the page (since $(A \circledcirc\!\!\!\!- B) \wedge (B \circledcirc\!\!\!\!\rightarrow A)$). Also in the example, C's content is not included because $C \not\circledcirc\!\!\!\!\rightarrow A$ (C returns `NO` in response to a request for `/soma-approval`). D's content is not included because $A \not\circledcirc\!\!\!\!- D$ (D is not listed in A's manifest). C returning $C \not\circledcirc\!\!\!\!\rightarrow A$ prevents pages from A accessing content from C in any way (including embedding content or performing cross-site request attacks). $A \not\circledcirc\!\!\!\!- D$ prevents web pages from A interacting with D in any way.

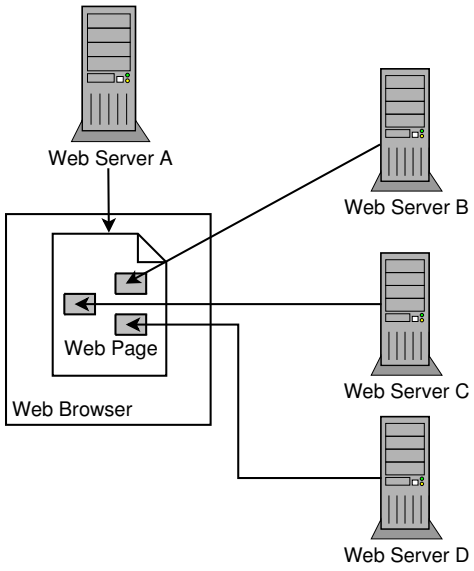

Figure 4: Inclusions allowed by the same origin policy

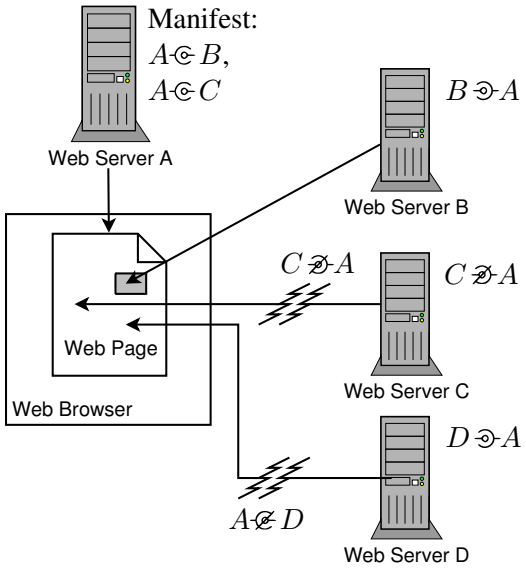

Figure 5: Inclusions allowed within the same origin mutual approval policy

In the example, A's web pages are trying to use content without C's approval [1], or A's web pages may be attempting a cross-site request forgery against C. In either case, the browser does not allow the communication.

In the case of content inclusions from D, the page is trying to include content but the manifest for A does not include D. The content from D is thus not loaded and not included (the web browser never checks to see if D would have granted approval or not). In this fashion SOMA prevents information from being

[1] Such inclusions may be considered stealing, either of the content itself or of the bandwidth needed to load the content.

sent to or received from an untrusted server.

3.5 Process of approval

The process the browser goes through when fetching content is described in Figure 6. First, the web browser gets the page from server A. In parallel, the browser retrieves the manifest file from server A using the same protocol (i.e. if the page is served over HTTPS, then the manifest will be retrieved over HTTPS as well). In this example, the web page requires content from web server C, so the browser first checks to see if C is in A's manifest. If $A \not\multimap C$, then the content is not loaded. If $A \multimap C$, then the browser verifies C's reciprocal approval by checking the `/soma-approval` details on C (again using the same protocol as the pending content request). If $C \not\multimap A$ then the browser again refuses to load the content. If $C \multimap A$ then the browser gets any necessary content from C and inserts it into the web page.

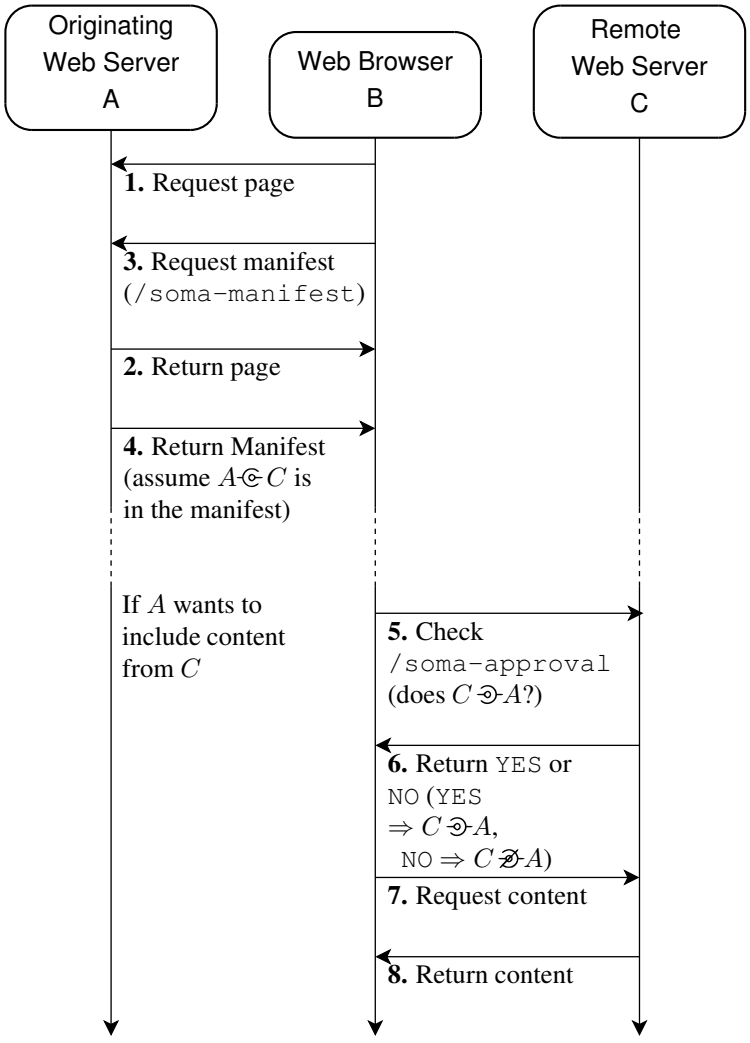

Figure 6: The mutual approval procedure

3.6 Compatibility with existing sites

In order to avoid breaking current web pages, SOMA defaults to a permissive mode if the manifest or approval files do not exist. These defaults reflect current web page behavior where all inclusions are allowed.

1. If the `soma-manifest` file does not exist on the origin, all inclusions are considered to be permitted by the origin site.
2. If the content provider has no `soma-approval` file, then any site is allowed to include content from this provider. In other words, the default `soma-approval` is `YES` if no file exists.

Note that these checks are independent, i.e., the lack of a `soma-manifest` does not prevent the loading of a `soma-approval` file and vice-versa.

4 Design Evaluation

4.1 Security Benefits

SOMA constrains JavaScript's ability to communicate by limiting it to mutually approved domains. Since many attacks rely upon JavaScript's ability to communicate with arbitrary domains, this curtails many types of exploitive activity in web browsers. Whereas currently *any* web server can be used to host malicious JavaScript or to receive stolen information, the list of potential attackers is narrowed significantly, either to insiders at the web site in question, or to one of its approved partners. As we explain below, this change would provide substantial additional protection in practice.

One key factor making SOMA a feasible defense is that the costs of implementation and operation are borne by those parties who stand the most to benefit and who are most suited to bear its costs. It also helps those who wish more control over what sites embed their content.

4.1.1 Recursive Script Inclusion

Script inclusion is only allowed from mutually approved domains. Therefore, if a script is included recursively, it still needs to come from a mutually approved domain, regardless of which domain included it. The use of the manifest to constrain inclusions means that attackers will no longer be able to store attack code on external domains unless they are in the manifest and mutually approved. Many current attacks rely on the ability to store code externally [24], therefore SOMA will force attackers to use new attack strategies.

4.1.2 Unrestricted outbound communication

Outbound communication under SOMA is controlled so that (explicit) information can only flow to and from mutually approved partners. Thus, attacker who wish to get information from a page now cannot have the browser send it to any arbitrary web server. This change blocks many existing cookie-stealing and similar information theft attacks, forcing attackers to compromise an approved partner in order to get such information.

While SOMA provides no protection against local covert communications channels, it does protect against most timing attacks based upon cached content [11], simply because with SOMA the attacker's website will in general not be approved by the victim's for content inclusion.

4.1.3 Cross-site request forgery

Cross-site request forgery attacks occur when a malicious web site causes a URL to be loaded from another, victim web site. SOMA dictates that URLs can only be loaded if a site has been mutually approved, which means that sites are only vulnerable to cross-site request forgery from sites on their approval lists. Specifically, the approval files limit the possible attack vectors for a cross-site request forgery attack, while the manifest file ensures that an origin site cannot be used in an attack on another arbitrary site.

SOMA thus allows a new approach to protect web applications from cross-site request forgery. Any page which performs an action when loaded could be placed on a subdomain (by the server operator) which grants approval only to trusted domains, such as those they control. This change would limit attacks to cases where the user has been fooled into clicking on a link. It is unlikely that sites will need to grant external access to action-causing scripts: even voting sites, which generally want to make it easy for people to vote from an external domain with just a click, usually include some sort of click-through to prevent vote fraud.

SOMA also leaks less information to sites than the current `Referer` HTML header (which is also sometimes used to prevent cross-site request forgery [22]). Because the `Referer` header contains the complete URL (and not just the domain), sensitive information can currently be leaked [19]. Many have already realized the privacy concerns related to the `Referer` URL and have implemented measures to block or change this header [35, 30]. These proposals also prevent current cross-site request forgery detection attempts; however, they do not conflict with SOMA.

4.1.4 Cross-site scripting

SOMA blocks the "cross" part of cross-site scripting, since information can no longer be loaded from or sent to external, unapproved domains. This change forces attackers to either compromise the targeted origin host or one of its mutually approved partners, or to inject their entire attack code into the web page, thereby increasing their chances of detection. In fact, since the code needed to mount many attacks is of significant size (e.g., setting appropriate style attributes as camouflage), when combined with SOMA, simple length restrictions already in place on some user content may be sufficient to prevent many attacks.

Even if attack code manages to load, its communication channels are limited. Many attacks require that information such as credit card numbers be sent to the attacker for later use, but this will no longer be possible with SOMA. Other attacks require the user to load dangerous content hosted externally, and these would also fail.

Thus, while some forms of cross-site scripting attacks remain viable, they are limited to attacks on the existing page that do not require communication through the browser to other non-approved domains. For example, it is not possible to steal cookies if there is no way to send the cookie information out to the attacker. It is possible that the site itself could provide the way (for example, cookies could be emailed out of a compromised webmail client or posted on a blog). Or, the attacker could instead choose to deface the page, since this attack requires the script only to modify the page. However, without the cross-site component, the remaining attacks are just single site code injection attacks, not cross-site attacks.

4.1.5 Bandwidth stealing

SOMA allows content providers more control over who uses their content. Thus SOMA offers a new way to prevent "bandwidth theft" where someone is including images or other content from a (non-consenting) content provider into their page using a direct link to the original file. Existing techniques usually require

the web server to verify the HTTP referer header, which can be problematic (as discussed in Section 4.1.3). SOMA provides a technique to do the verification in the browser, not relying on HTTP referer.

Also known as hotlinking or inline linking, bandwidth theft is used maliciously by phishing sites, but may also be used unintentionally by people who do not know better. Regardless of the intent, this can still be damaging. While the content provider is paying hosting costs associated with serving up that file, it may be pulled in by, for example, a very popular blog or aggregate site that would generate a huge number of additional views. At the extreme, this could result in the content provider exceeding their bandwidth cap and being charged extra hosting fees or having their site shut down.

Many smaller sites would rather their content be used only by them for visitors to their site, and SOMA allows them to specify this and have browsers enforce this behavior. Bandwidth theft is often performed by people who are simply unaware that this is inappropriate behavior [6], and SOMA can address this since doing the wrong thing will simply not work.

4.2 Incremental Deployment

SOMA is designed to gracefully handle sites which are unaware of SOMA or have not yet been configured. More specifically, if the `soma-manifest` and/or `soma-approval` files do not exist (or do not contain SOMA specific identifying strings), the browser defaults to current permissive behavior, that is, assumes that inclusions are allowed. Thus, a SOMA-enabled browser can run on current web pages without any difference in behavior.

If only the origin site has a `soma-manifest`, then SOMA still provides partial security coverage, enforcing the policy that is defined in the `soma-manifest`. If the origin site does not have a manifest file, but the content provider gives approval information through `soma-approval` then the policy defined by the content provider is enforced.

In order to verify that files returned in response to requests for `soma-manifest` and `soma-approval` are related to SOMA, we stipulate that the first line of the `soma-manifest` file must contain `SOMA Manifest` and the `soma-approval` file must contain only the word `YES` or `NO`. This is necessary since many websites return a generic page even when the request has not been found, and this must not be confused with intentional responses to SOMA requests.

The full benefits of SOMA are available when origins and content providers both provide SOMA-related files, but the design is such that it is possible for either side to start providing files without needing extensive coordination to ensure that both are provided at the same time. In other words, incremental deployment is possible. In addition, even if one site refuses to provide policy files for whatever reason, others can still obtain lesser security guarantees. Moreover, the support of SOMA at servers need not be synchronized with deployment of SOMA at browsers.

A more security-oriented default policy would be possible, with SOMA assuming a `NO` response if the manifest or approval files are not found by the browser. This could potentially provide additional security even on sites which do not provide policy, as well as encouraging sites which do not have policies to set them. However, it would break almost all existing web pages, almost surely preventing the adoption of SOMA. The permissive default was chosen to reflect current browser behavior and to make it easier for SOMA to be deployed.

4.3 Deployment Costs

The browser, the origin sites, and content inclusion provider sites all bear the costs in deploying SOMA. Note that unlike some solutions which rely heavily upon user knowledge (e.g. the NoScript add-on for

Mozilla Firefox [21]), SOMA requires no additional effort on the part of the user browsing the web site. Instead, policies are set by server operators, who are expected have more information about what constitutes good policy for their sites.

4.3.1 Deployment in the browser

The SOMA policy is enforced by the web browser, so changes are required within web browsers. We have created a prototype add-on for Mozilla Firefox 2 as discussed in greater detail in Section 5.

4.3.2 Deployment on origin sites

Each origin site which wishes to benefit from the protections of SOMA needs to provide a `soma-manifest` file. This is a text file which contains a list of content-providing sites from which the origin wishes to include content. As mentioned earlier, these content providers are specified by domain name, protocol and (optionally) port.

This list can be determined by looking at pages on the site and compiling a list of content providers. This could be automated using a web crawler, or done by an admin who is willing to set policy. (It is possible that sites will wish to set more restrictive policy than the site's current behavior.) We examined the main page on popular sites to determine the approximate complexity of manifests required, and these results are detailed in Section 5.5.2.

4.3.3 Deployment on content provider sites

Content providers wishing to take advantage of SOMA need to provide either a file or script which can handle requests to `soma-approval`. The time needed to create this policy script depends heavily upon the needs of the site in question, and may range from a simple yes-to-all or no-to-all to more complex policies based upon client relationships. Fortunately, simple policies are likely to be desired by smaller sites (which are unlikely to have the resources to create complex policies), and complex policies are likely to be required only by sites who have the necessary expertise.

Many sites will not wish to be external content providers and their needs will be easily served by a `soma-approval` file that just contains `NO`. Such a configuration will be common on smaller sites such as personal blogs. It will also be common on high-security sites such as banks, who want to be especially careful to avoid XSRF and having their images used by phishing sites. Phishing sites will have to copy over images, facilitating legal action over copyright violations.

Other sites may wish to be content providers to everyone. Sites such as Flickr and YouTube that wish to allow all users to include content will probably want to have a simple `YES` policy. This is most easily achieved by simply not hosting a `soma-approval` file (as this is the default), or by creating one that contains the word `YES`.

The sites requiring the most configuration are those who want to allow some content inclusions rather than all or none. For example, advertisers might want to provide code to sites displaying their ads. The list of domains that need to be approved is already maintained, as this is part of their client list. This database could then be queried to generate the approval list. Or a company with several web applications might want to keep them on separate domains but still allow interaction between them. Again, the necessary inclusions will be known in advance and necessary policy could be created by a system administrator or web developer.

For an evaluation of the performance impact of SOMA, see Section 5.5.3.

4.4 Limitations

SOMA is designed to improve the same origin policy by imposing further constraints upon external inclusions and thus external communications. As such, it does not prevent attacks that do not require external communications or external code inclusions. Note that in current attack code, outside communication is frequently used [24].

SOMA does not stop attacks to or from mutually approved partners. In order to avoid these attacks, it would be necessary to impose finer-grained control or additional separation between components. This sort of protection can be provided by the mashup solutions described in Section 6, albeit at the cost of extensive and often complex web site modifications.

SOMA cannot stop attacks on the origin where the entire attack code is injected, if no outside communication is needed for the attack. This could be web page defacement, same-site request forgery, or sandbox-breaking attacks intended for the user's machine. Some complex attacks might be stopped by size restrictions on uploaded content. More subtle attacks might need to be caught by heuristics used to detect cross-site scripting. Some of these solutions are described in Section 6.

SOMA cannot stop attacks from malicious servers not including content from remote domains. These would include phishing attacks where the legitimate server is not involved.

5 Prototype

5.1 Description

In order to test SOMA, we created an add-on for Mozilla Firefox 2.0. It can be installed in an unmodified installation of Mozilla Firefox the same way as any other add-on: the user clicks an installation link and is prompted to confirm the install. If they click the install button, the add-on is installed and begins to function after a browser restart.

The SOMA add-on provides a component that does the necessary verification of the `soma-manifest` and `soma-approval` files before content is loaded.

Since it is not possible to insert policy files onto sites over which we had no control, we used a proxy server to simulate the presence of manifest and approval files on popular sites.

5.2 Performance

The primary overhead in running SOMA is due to the additional latency introduced by having to request a `soma-manifest` or `soma-approval` from each domain referenced on a web page. While these responses can be cached (like other web requests), the initial load time for a page is increased by the time required to complete these requests. Because the manifest can be loaded in parallel with the origin page (subsequent requests can not be sent until the browser has received and parsed the origin page anyway), we do not believe manifest load times will affect total page load times. Because `soma-approval` files must retrieved before contacting other servers, however, overhead in requesting them will increase page load times.

Because sites do not currently implement SOMA, we estimate SOMA's overhead using observed web request times. First, we determined the average HTTP request round-trip time for each of 40 representative web sites[2] on a per-domain basis using PageStats [9]. We used this per-domain average as a proxy for the time to retrieve a `soma-approval` from a given domain. Then, to calculate page load times using SOMA,

[2]Our representative sample included banks, news sites, web e-mail, e-commerce, social networking, and less popular sites.

we increase the time to request all content from each accessed domain by the `soma-approval` request estimate for that domain. The time of the last response from any domain then serves as our final page load time.

After running our test 30 times on 40 different web pages, we found that the average additional network latency overhead due to SOMA increased page load time from 2.9 to 3.3 seconds (or 13.28%) on non-cached page loads. On cached page loads, our overhead is negligible (since `soma-approval` is cached). We note that this increase in latency is due to network latency and not CPU usage. If we assume that 58% of page loads are revisits [32], the average network latency overhead of SOMA drops to 5.58%.

Because `soma-approval` responses are extremely small (see Section 5.5.3), they should be faster to retrieve than the average round-trip time estimate used in our experiments. Thus, these values should be seen as a worst-case scenario; in practice, we expect SOMA's overhead to be significantly less.

5.3 Compatibility with Existing Web pages

To test compatibility with existing web pages, the global top 45 sites as ranked by Alexa [3] were visited in the browser with and without the SOMA add-on. No SOMA compatibility issues were detected in these tests. In addition, one author ran the SOMA add-on for 2 weeks while doing regular browsing, and no SOMA incompatibilities were observed. These results were expected, as SOMA was designed for compatibility and incremental deployment.

5.4 Attacks

In order to verify that SOMA actively blocks information leakage, cross-site request forgery, cross-site scripting, and content stealing, we created examples of these attacks. We specifically tested the following attacks with the SOMA add-on:

1. A GET request for an image on another web site (testing both GET based XSRF as well as content stealing).
2. A POST request to a page on another web site done through JavaScript (testing POST based XSRF).
3. An iframe inclusion from another web site (testing iframe injection based XSS).
4. Sending either a cookie or personal information to another web site (testing information leakage).
5. A script inclusion from another web site (testing XSS injection).

All attacks were hosted at domain A and used domain B as the other domain involved. All attacks were successful without SOMA and we found that with SOMA either a manifest at domain A not listing B or a `soma-approval` at domain B which returned `NO` for domain A prevented the attacks.

5.5 Deployment Costs

5.5.1 Browser: SOMA Add-on

The SOMA add-on, when prepared into the standard XPI package format used by Mozilla Firefox, is 7kB. Uncompressed, the entire add-on is 25kB. The component which does the actual SOMA mutual approval process is 21kB; the rest is installation files and chrome so that the browser provides a visual indication that the add-on is loaded.

5.5.2 Origin sites: Manifest files

To determine approximate sizes for manifests, we used the PageStats add-on [9] to load the home page for the global top 500 sites as reported by Alexa [3] and examined the resulting log, which contains information about each request that was made. On average, each site requested content from 5.45 domains other than the one being loaded, with a standard deviation of 5.3. The maximum number of content providers was 32 and the minimum was 0 (for sites that only load from their own domain).

Of course, a site's home page may not be representative of its entire contents. So, as a further test we traversed large sections of a major news site and determined that the number of domains needed in the manifest was approximately 45; this value was close to the 33 needed for the site's home page.

Given the remarkable diversity of the Internet, there probably exist sites today that would require extremely large manifest files. This cursory survey, however, gives evidence that manifests for common sites would be relatively small.

5.5.3 Content provider sites: Approval files

Approvals result in tiny amounts of data being transferred: either a `YES` or `NO` response (around 4 bytes of data) plus any necessary headers.

Using data from the top 500 Alexa sites [3], we examined 3244 cases in which a content provider served data to an origin site. The average request size was 10459 bytes. Because many content providers are serving up large video, however, the standard deviation was fairly large: 118197 bytes. The median of 2528 bytes is much lower than the average. However, even this smaller median dwarfs the 4 bytes required for a `soma-approval` response. As such, we feel it safe to say that the additional network load on content providers due to SOMA is negligible compared to the data they are already providing to a given origin site.

6 Related Work

Web-based execution environments have all been built with the understanding that unfettered remote code execution is extremely dangerous. SSL and TLS can protect communication privacy, integrity, and authenticity; code signing [27, 31] can prevent the execution of unauthorized code; neither, however, protect against the execution of malicious code within the browser. Java [8] was the first web execution environment to employ an execution sandbox [34] and the same origin policy for restricting network communication. Subsequent systems for executing code within a browser, including JavaScript, have largely followed the model as originally embodied in Java applets.

While there has been considerable work on mitigating the failures of language-based sandboxing [17] and on sandboxing other, less trusted code such as browser plugins and helper applications [12], only recently have researchers begun addressing the limitations of sandboxing and same origin policy with respect to JavaScript applications.

Many researchers have attempted to detect and block malicious JavaScript. Some have proposed to instrument JavaScript automatically to detect known vulnerabilities [26], while others have proposed to filter JavaScript to prevent XSS [18] and XSRF [16] attacks. Another approach has been to perform dynamic taint tracking (combined with static analysis) to detect the information flows associated with XSS attacks [33]. Instead of attempting to detect dangerous JavaScript code behavior before it can compromise user data, SOMA prevents the unauthorized cross-domain information flows using site-specific policies.

Recently several researchers have focused on the problem of web mashups. Mashups are composite JavaScript-based web applications that draw functionality and content from multiple sources. To make

mashups work within the confines of same origin policy, remote content must either be separated into separate iframes or all code must be loaded into the same execution context. The former solution is in general too restrictive while the latter is too permissive; mashup solutions are designed to bridge this gap. Two pioneering works in this space are Subspace [15] and MashupOS [14].

SOMA prevents the creation of mashups using unauthorized code, i.e., in order for a mashup to work with SOMA, every web site involved in it must explicitly allow participation. While such restrictions may inhibit the creation of novel, third party mashup applications, they also prevent attackers from creating malicious mashups (e.g., combinations of a legitimate bank's login page and a malicious login box). Given the state of security on the modern web, we believe such a trade-off is beneficial and, moreover, necessary.

While the general problem of unauthorized information flow is a classic problem in computer security [10], little attention has been paid in the research community to the problems of unauthorized cross-domain information flow in web applications beyond the strictures of same origin policy—this, despite the fact that XSS and XSRF attacks very heavily rely upon such unauthorized flows. Of course, the web was originally designed to make it easy to embed content from arbitrary sources. With SOMA, we are simply advocating that any such inclusions should be approved by both parties.

While SOMA is a novel proposal, we based the design of `soma-approval` and `soma-manifest` on existing systems. The `soma-approval` mechanism was inspired by the `crossdomain.xml` [4] mechanism of Flash. External content may be included Flash applications only from servers with a `crossdomain.xml` file [4] that lists the Flash applications' originating server. Because the response logic behind a `soma-approval` request can be arbitrarily complex, we have chosen to specify that it be a server-side script rather than an XML file that must be parsed by a web browser.

The `soma-manifest` file was inspired by Tahoma [7], an experimental VM-based system for securing web applications. Tahoma allows users to download virtual machine images from arbitrary servers. To prevent these virtual machines from contacting unauthorized servers (e.g., when a virtual machine has been compromised), Tahoma requires every VM image to include a manifest specifying what remote sites that VM may communicate with.

Note that by themselves Flash's `crossdomain.xml` and Tahoma's server manifest do not provide the type of protection provided by SOMA. With Flash, a malicious content provider can always specify a `crossdomain.xml` file that would allow a compromised Flash program to send sensitive information to the attacker. With Tahoma, a malicious origin server can specify a manifest that would cause a user's browser to send data to an arbitrary web site, thus causing a denial-of-service attack or worse. By including both mechanisms, we address the limitations of each.

7 Discussion and Conclusion

Most JavaScript-based attacks require that compromised web pages communicate with attacker-controlled web servers. Here we propose an extension to same origin policy—the same origin mutual approval (SOMA) policy—which restricts cross-domain communication to a web page's originating server and other servers that mutually approve of the cross-site communication. By preventing inappropriate or unauthorized cross-domain communication, attacks such as cross-site scripting and cross-site request forgery can be blocked.

The SOMA architecture's benefits versus other JavaScript defenses include: 1) it is incrementally deployable with incremental benefit; 2) it imposes no configuration or usage burden on end users; 3) the required changes in browser functionality and server configuration affect those who have the most reason to be concerned about security, namely the administrators of sensitive web servers and web browser developers; 4) these changes are easy to understand, simple to implement technically, and efficient in execution;

and 5) it gives server operators a chance to specify what sites can interact with their content. While SOMA does not prevent attackers from injecting JavaScript code, with SOMA such code cannot leak information to attackers without going through an approved server.

We believe that SOMA represents a reasonable and practical compromise between benefits (increased security) and costs (adoption pain). Perhaps more significantly, our proposal of the SOMA architecture highlights that the ability to create web pages using arbitrary remote resources is a key enabling factor in web security exploits (including some techniques used in phishing). While other JavaScript defenses will no doubt arise, we believe that among the contributions of this paper are a focus on the underlying problem, namely, deficiencies in the JavaScript same origin policy, and the identification of several important characteristics of a viable solution.

It is easy to dismiss any proposal requiring changes to web infrastructure; however, there is precedence for wide scale changes to improve security. Indeed, much as open email relays had to be restricted to mitigate spam, we believe that arbitrary content inclusions must be restricted to mitigate cross-site scripting and cross-site request forgery attacks. We hope this insight helps clarify the threats that must be considered when creating next-generation web technologies and other Internet-based distributed applications.

References

[1] CERT® advisory CA-2000-02 malicious HTML tags embedded in client web requests. Technical Report CA-2000-02, CERT, 2000.

[2] The cross site scripting (XSS) FAQ. Web Page (viewed 14 Apr 2008), August 2003. `http://www.cgisecurity.com/articles/xss-faq.shtml`.

[3] Alexa top 500 sites. Web Page (viewed 14 Apr 2008), April 9 2008. `http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none`.

[4] Adobe Systems Incorporated. External data not accessible outside a Macromedia Flash movie's domain. Technical Report tn_14213, Adobe Systems Incorporated, February 2006.

[5] R. Auger. The cross-site request forgery (CSRF/XSRF) FAQ. Web Page (viewed 14 Apr 2008), January 2007. `http://www.cgisecurity.com/articles/csrf-faq.shtml`.

[6] R. Berends. Bandwidth stealing. *website-awards.net*, April 1 2001.

[7] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *Proc. IEEE Symposium on Security and Privacy*, pages 350–364, 2006.

[8] D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. *sp*, 00:0190, 1996.

[9] S. DeDeo. Pagestats. Web Page (viewed 14 Apr 2008), May 2006. `http://www.cs.wpi.edu/~cew/pagestats/`.

[10] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(2):236–243, 1976.

[11] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Proc. 7th ACM conference on Computer and Communications Security*, pages 25–32, 2000.

[12] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proc. 6th Usenix Security Symposium*, 1996.

[13] J. Grossman and T. Niedzialkowski. Hacking intranet websites from the outside – JavaScript malware just got a lot more dangerous. In *Blackhat USA*, Aug 2006.

[14] J. Howell, C. Jackson, H. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[15] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the 16th international conference on World Wide Web*, pages 611 – 620. Association for Computing Machinery, 2007.

[16] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *2nd IEEE Communications Society International Conference on Security and Privacy in Communication Networks (SecureComm)*, Baltimore, MD, August 2006.

[17] K. Keahey, K. Doering, and I. Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *Proc. Fifth IEEE/ACM International Workshop on Grid Computing*, pages 34–42, 2004.

[18] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *The 21st ACM Symposium on Applied Computing (SAC 2006), Security Track*, Dijon, France, April 2006.

[19] J. Kyrnin. Are you invading your customers' privacy? Web Page (viewed 14 Apr 2008). `http://webdesign.about.com/od/privacy/a/aa112601a.htm`.

[20] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 221–234, New York, NY, USA, 2006. ACM Press.

[21] G. Maone. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience! Web Page (viewed 14 Apr 2008), November 2007. `http://noscript.net/`.

[22] A. D. Miglio. "Referer" field used in the battle against online fraud. Web Page (viewed 14 Apr 2008), Jan 2008. `http://www.symantec.com/enterprise/security_response/weblog/2008/01/referer_field_used_in_the_batt.html`.

[23] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. Technical Report provos-2008a, Google, February 4 2008.

[24] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots '07)*, 2007.

[25] J. Reimer. Microsoft apologizes for serving malware. *Ars Technica*, February 21 2007.

[26] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[27] A. Rubin and D. Geer. Mobile code security. *IEEE Journal on Internet Computing*, 2(6):30–34, 1998.

[28] J. Ruderman. The same origin policy. Technical report, Mozilla, 2001.

[29] B. Schiffman. Rogue anti-virus slimeballs hide malware in ads. *Wired*, November 15 2007.

[30] T. Scott. Smarter image hotlinking prevention. *A List Apart*, April 13 2004.

[31] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-carrying code (MCC): a new paradigm for mobile-code security. In *Proc. 2001 workshop on New security paradigms*, pages 23–30, Sep 2001.

[32] L. Tauscher and S. Greenberg. How people revisit web pages: empirical findings and implications for the design of history systems. In *International Journal of Human-Computer Studies*, 1997.

[33] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, San Diego, CA, February 2007.

[34] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SIGOPS Operating System Review*, 27(5):203–216, 1993.

[35] WordPress.org. Enable sending referrers. Web Page (viewed 14 Apr 2008). `http://codex.wordpress.org/Enable_Sending_Referrers`.