View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Syracuse University Research Facility and Collaborative Environment

Syracuse University
SURFACE

Electrical Engineering and Computer Science   College of Engineering and Computer Science

2009

ESCUDO: A Fine-grained Protection Model for Web Browsers

Karthick Jayaraman
Syracuse University

Wenliang Du
Syracuse University, wedu@syr.edu

Balamurugan Rajagopalan
Syracuse University

Steve J. Chapin
Syracuse University

Follow this and additional works at: https://surface.syr.edu/eecs

Part of the Computer Engineering Commons

Recommended Citation

Jayaraman, Karthick; Du, Wenliang; Rajagopalan, Balamurugan; and Chapin, Steve J., "ESCUDO: A Fine-grained Protection Model for Web Browsers" (2009). *Electrical Engineering and Computer Science*. 5.
https://surface.syr.edu/eecs/5

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Department of Electrical Engineering and Computer Science

Technical Report

SYR-EECS-2009-01 Sep 01, 2009

ESCUDO: A Fine-grained Protection Model for Web Browsers

Karthick Jayaraman kjayaram@syr.edu
Wenliang Du wedu@syr.edu
Balamurugan Rajagopalan barajago@syr.edu
Steve J. Chapin chapin@syr.edu

ABSTRACT: Web applications are no longer simple hyperlinked documents. They have progressively evolved to become highly complex---web pages combine content from several sources (with varying levels of trustworthiness), and incorporate significant portions of client-side code. However, the prevailing web protection model, the *same-origin policy*, has not adequately evolved to manage the security consequences of this additional complexity. As a result, web applications have become attractive targets of exploitation. We argue that this disconnection between the protection needs of modern web applications and the protection models used by web browsers that manage those applications amounts to a failure of access control. In this paper, we present Escudo, a new web browser protection model designed based on established principles of mandatory access control. We describe our implementation of a prototype of Escudo in the Lobo web browser, and illustrate how web applications can use Escudo for securing their resources. Our evaluation results indicate that Escudo incurs low overhead. To support backwards compatibility, Escudo defaults to the same-origin policy for legacy applications.

KEYWORDS: Web browsers, access control, web security

Syracuse University - Department of EECS,
4-206 CST, Syracuse, NY 13244
(P) 315.443.2652 (F) 315.443.2583
http://eecs.syr.edu

ESCUDO: A Fine-grained Protection Model for Web Browsers

Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J. Chapin
Department of EECS, Syracuse University
{kjayaram,wedu,barajago,chapin}@syr.edu

Abstract

Web applications are no longer simple hyperlinked documents. They have progressively evolved to become highly complex—web pages combine content from several sources (with varying levels of trustworthiness), and incorporate significant portions of client-side code. However, the prevailing web protection model, the *same-origin policy*, has not adequately evolved to manage the security consequences of this additional complexity. As a result, web applications have become attractive targets of exploitation. We argue that this disconnection between the protection needs of modern web applications and the protection models used by web browsers that manage those applications amounts to a failure of access control. In this paper, we present ESCUDO, a new web browser protection model designed based on established principles of mandatory access control. We describe our implementation of a prototype of ESCUDO in the Lobo web browser, and illustrate how web applications can use ESCUDO for securing their resources. Our evaluation results indicate that ESCUDO incurs low overhead. To support backwards compatibility, ESCUDO defaults to the same-origin policy for legacy applications.

1 Introduction

Initially, web applications comprised a set of documents that mostly contained text to be rendered and hyperlinks to other documents, with little or no client-side code. All the content originated from a single, trusted source. Over the last several years, in the race to add interactive features, web applications have progressively become more complex. In more recent times, web applications have evolved to become highly interactive applications that execute on both the server and client. As a result, web pages in modern applications are no longer simple documents–they now comprise highly dynamic contents that interact with each other. In some sense, a web page has now become a "system"–the dynamic contents are programs running in the system, and they interact with users, access other contents both on the web page and in the hosting browser, invoke browser APIs, and interact with programs on the server side.

Moreover, today's web pages no longer draw contents from a single source; contents are now derived from several sources with varying levels of trustworthiness. Contents may be included by the application itself, derived from user-supplied text, or from partially trusted third parties. During parsing, rendering, and execution inside the browser, the dynamic and static contents of web pages can both act and be acted upon by other entities—in classic security parlance, they can be instantiated as both principals and objects. These principals and objects are only as trustworthy as the sources from which they originate.

The security of a web application is primarily dependent on the integrity and confidentiality of its resources inside the web browser. For example, session identifiers in cookies need to be protected against access by untrusted principals; code from untrusted sources must be authorized before it is allowed to modify any trusted content on a web page. Without appropriate access control in web applications, we cannot preserve the trustworthiness of contents, and security could be compromised. If we consider each web page as a "system," we need an adequate protection model in browsers to mediate the interactions within such a system.

All web browsers implement a protection model called the *same-origin policy*. Unfortunately, this model has not adequately evolved to manage the security consequences of the additional complexity in modern web pages. It cannot distinguish gradations in trustworthiness, nor does it provide sufficient isolation between web browser objects to ensure proper access control. As a result, web applications have become attractive targets of exploitation. Both cross-site-scripting attacks and cross-site-request forgery attacks are examples of untrusted principals exercising control over trusted objects inside the web browser. We argue that the root cause of the problem is a failure of access control. The same-origin policy clearly violates two important principles of access control, namely separation of privilege and principle of least privilege [32].

Because of access-control failures, web applications that embed third party content in their web page cannot restrict the permissions of the third party code. For example, a blog publisher may sell a small portion of his web page to an advertising network. The advertising network, in turn, accepts Javascript ads from its clients and displays them on the publisher's web page. The publisher has no further control over what appears in that ad space—he trusts the network to have verified all content. An attacker posing as an advertiser could compromise the integrity of the publishers web application using a malicious JavaScript program [36]. JavaScript verifiers such as ADsafe [12] could be used by an advertisement network to verify a JavaScript program, but that does not change the publisher's position: he is relying on a third-party to vouch for the trustworthiness of Javascript programs that will run in his own web pages.

There have been other approaches for dealing with this access-control failure. Web applications, as a first line of defense, employ input validation and content filtering at the server when generating the web page. The objective of this step is preventing known attacks from instantiating an untrustworthy principal inside a web page. For example, to defeat cross-site scripting attacks, we can filter out all the code from contents originating from untrusted sources. This first-line of defense has proven to be difficult to implement properly; many vulnerabilities are because of the errors in such a process [15, 17]. Second, there are browser patches that address specific attacks [18]. In general, all these approaches address the symptoms of specific problems without addressing the fundamental root cause—the lack of a robust protection model suitable for modern web applications.

We describe an alternate approach that addresses the access-control failure in web browsers by redesigning the underlying access-control model, attacking the root of the problem. Redesigning the access-control model for web browsers involves four challenges. First, the access-control model should be able to identify principals and objects at required granularity. Second, the access-control model should use an appropriate policy to secure content with varying levels of trustworthiness. Third, a challenge unique to web applications is distributed enforcement–the applications at the server are aware of trustworthiness, but the actual interactions that have to be restricted happen at the browser. Finally, the new model should be backward compatible with the same-origin policy to facilitate incremental deployment.

In this paper, we describe ESCUDO, a fine-grained web browser protection model, based on vetted access-control principles to protect modern web applications. To the best of our knowledge, this is the first work on redesigning the access-control model for web browsers. ESCUDO is designed to enforce separation of privilege and the principle of least privilege, and to provide adequate granularity in both specification and enforcement. We argue that the protection requirements of web applications are similar to operating systems. Some operating systems use hierarchical protection rings (HPR) to enforce their protection requirements. ESCUDO is an adaptation of HPR tailored to meet the protection requirements of web applications.

To address the distributed enforcement problem, we describe a method that web applications could use to identify the principals, objects, and their trustworthiness and configure their resources at the granularity required by them to reflect their protection needs. The method is backward compatible with non-ESCUDO browsers, which ignore the configuration and default to the same-origin policy.

We implemented a prototype of ESCUDO for the Lobo browser and our evaluation results show that ESCUDO incurs around 5% overhead. We illustrate our experience in building web applications for ESCUDO using two open-source web applications as case studies. We modified two open source web applications to

use ESCUDO. We analyzed the web applications to understand their security requirements and configured them to use ESCUDO to enforce the security requirements. The key contributions of the paper can be summarized as follows:

- ESCUDO is a new fine-grained web browser protection model to meet the protection requirements of modern applications.
- A backward-compatible configuration method that web applications can use to identify the principals, objects, and their trustworthiness in order to use ESCUDO.
- A prototype implementation of ESCUDO on the Lobo web browser.
- Case studies illustrating our experience of building web applications for ESCUDO.

2 Protection Requirements for Web Applications

In this section, we will describe the protection requirements of web applications by providing an analysis of the principals, objects, and the characteristics of modern web applications. Finally, we describe the inadequacy of the same-origin policy in meeting the protection requirements.

2.1 Principals and Objects

In a web application, principals are action-inducing HTML excerpts such as JavaScript programs, and objects are application resources such as the web page contents and cookies that are targets of actions. Some HTML excerpts, such as JavaScript programs, may act as both principals and objects. Table 1 describes the principals and objects inside a web application.

HTTP-request Issuing Principals: HTTP-request issuing principals are HTML tags such as *a*, *img*, *form*, *embed*, and *iframe* that instruct the web browser to issue an HTTP request.

Script-invoking Principals: Script-invoking principals are HTML constructs such as *script* and the CSS *expression* that can invoke the JavaScript interpreter. Additionally, web applications can specify user-interface (UI) event handlers to be invoked for specific events using attributes such as `onload`, `onmouseover`, etc.

Plugins: Plugins are content-specific run-time environments for certain types of contents such as Flash, Silverlight, and PDF. Additionally, browsers such as Firefox provide a framework for creating extensions, enabling users to extend the functionality of the browser. Plugins and extensions have their own security models and may or may not be controlled by the web applications. In this paper, we only focus on the principals that can be controlled by the web applications.

Document object model (DOM): Internally, web browsers represent the contents of a web page using a data structured called the DOM, in which all the HTML tags and their content are organized in a hierarchical fashion. Each HTML tag is a DOM element. DOM elements have a special feature—they can act as both principals and objects. Some DOM elements are script-invoking principals or HTTP-request initiating principals. Such DOM elements act as principals momentarily when they are instantiated. On the other hand, they act as objects when they are the targets of modification via the DOM API.

Principals	Objects
HTTP-request issuing principals - HTML Form - HTML Anchor - HTML Img - HTML Iframe - HTML Emded **Script-invoking principals** - JavaScript Programs - UI event Handlers **Plugins** (Cannot be controlled by web applications)	**Document object model (DOM)** **Cookies** **Native Code API** -XMLHttpRequest API -DOM API **Browser State** - History - Visited link information

Table 1: Principals and objects inside the web browser.

Cookies: Web applications create cookies in web browsers; cookies typically contain data used to track sessions. After a cookie is created, web browsers attach the cookies in all subsequent HTTP requests back to the web application. Therefore, cookies are objects that are implicitly accessed in all HTTP requests. In addition, JavaScript programs may explicitly manipulate cookies.

Native Code: Native browser code is exposed to JavaScript programs via an API. For example, the `XMLHttpRequest` API is an example of native code that helps JavaScript programs interact with server-side programs. Similarly, the DOM API is used by JavaScript programs to access and modify the web page. Web applications may not want to expose these API to untrusted code. Therefore, the ability to invoke such API should be a controllable privilege.

Browser State: Web browsers maintain browsing history and visited link information for each browsing session with a web site. This information is part of the state of a browsing session and is accessible to JavaScript programs through the DOM API. Browsing history is a log of recently visited URL and users may use this information to instruct the web browser to render a previously visited web page. Visited link information is used by web browsers to differentiate recently visited from unvisited URL—typically, web browsers use differing colors to display visited and unvisited links.

2.2 Protection Needs

We outline two characteristics of modern applications that are relevant for motivating our protection requirements:

Increasing Use of Client-side Code: Earlier, web applications primarily executed on the server and only web pages were delivered to browsers. With the introduction of JavaScript programs, web applications could additionally execute in the browser to provide some interactive features. JavaScript programs are commonly used to display drop-down menus by updating the contents of the web page. Furthermore, AJAX enables JavaScript programs to communicate with the application at the server. An instant-messaging application might use an AJAX-based JavaScript program for communicating with the server and updating the chat window.

Content with Varying Levels of Trustworthiness: In modern applications, the content inside web pages is derived from multiple sources with nonuniform trustworthiness. Therefore, content inside web pages is no longer only trusted and provided by the application itself. There are several examples of applications including untrusted content. Blogs and wikis enable users to provide arbitrary text that will be part of the web pages. Because the text is supplied by the user, it should not be trusted. There are also examples of applications including semi-trusted content. An online auction application may enable a seller to create a web page in its application and may also allow the seller to provide JavaScript programs in the page to enable some rich functionality. A social networking application may allow users to add applications, essentially JavaScript programs, in their profile to extend the functionality of their profile pages. Web applications frequently add third party JavaScript programs for adding some features. For example, an application may include third party JavaScript program for keeping track of site statistics. Online advertising that we discussed earlier is another example of including semi-trusted content.

As a direct consequence of these two characteristics, we have principals of varying trustworthiness inside the web page. Currently, these principals access or modify content in the web page, invoke native API, and communicate with the application at the server, irrespective of their trustworthiness. Saltzer and Schroeder [32] have summarized eight design principles for building protection mechanisms: economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability. Of the eight guidelines, the same-origin policy clearly violates two principles, namely least privilege and separation of privilege, but has done a fairly good job with respect to the other characteristics. The same-origin policy's non-conformance with sound design principles leads directly to its failure to meet the protection needs of modern applications. Based on our analysis of modern applications and vetted principles, a protection model for web browsers requires three characteristics:

1. **Separation of Privilege:** Separation of privilege indicates that, if possible, privileges in a system should be divided into less powerful privileges, such that no single accident, deception, or breach of trust is sufficient to compromise the protected information.

2. **Principle of Least Privilege:** The protection model should be able to limit the interactions of principals based on their trustworthiness. Essentially, a principal should not have more privileges to access information or resources than required for its legitimate purpose. In addition, a principal should not be able to elevate its privilege in an uncontrolled manner.

3. **Fine Granularity:** The protection model should be able to identify principals at a sufficient granularity to ascertain their trustworthiness. Therefore, origins alone are insufficient for this purpose. Having fine granularity is essential for achieving the principle of least privileges.

2.3 The Same-Origin Policy is Inadequate

The same-origin policy (SOP) identifies an application's origin as a unique combination of ⟨*protocol, domain, port*⟩. For instance, `http://www.amazon.com/index.php` and `http://www.amazon.com/search.php` belong to the same origin, but `http://www.gmail.com` and `http://www.amazon.com` do not belong to the same origin because they have differing domains. Similarly, `http://www.gmail.com` and `https://www.gmail.com` do not belong to the same origin because they use different protocols. Web browsers associate application resources such as cookies and document object model (DOM) to their origin, and the SOP prevents JavaScript programs from one origin from accessing application resources belonging to other origins.

Under the SOP, all principals inside the web application are associated with a single principal identified by the origin and are associated with all the privileges irrespective of their trustworthiness, violating the

principle of least privilege. In addition, principals and resources across applications are not appropriately isolated from one another. Both cross-site-scripting (XSS) attacks and cross-site-request forgery (CSRF) attacks are a side effect of these inadequacies.

In XSS attacks, an attacker deftly constructs input data for an application that is interpreted as JavaScript by the web browser and executes with all privileges. Ideally, the JavaScript program should execute with limited or no privileges because it was derived from untrusted web content.

In CSRF attacks, a malicious site forges and injects a request into a victim user's active session with a trusted site. Some HTML tags such as *a*, *img*, and *iframes* can initiate an HTTP request. There are no restrictions on the URL that can be used in these HTML tags. In addition, web browsers automatically attach the target site's cookies to the HTTP request, irrespective of who is making the request. A malicious site abuses this weakness to forge a request for a trusted site. Ideally, principals and objects across applications should be isolated from these types of unintended interferences.

3 Our Approach

We need fundamental changes to the existing web browser protection model to address the protection needs of modern applications. Our approach is to design a web browser protection model based on vetted mandatory access-control (MAC) principles. In our proposed model, developers can configure their application by appropriately specifying the principals, objects, and their trustworthiness. Web applications communicate the configuration to the web browser, where the proposed access-control model enforces access decisions based on the configuration. This is typical of any MAC system, where a system administrator configures the system and system-level mechanisms enforce access decisions based on the configuration [8].

Conceptually, access control is the ability to decide who can do what to whom in a system. An access-control system consists of three components: principals, objects, and an access-control model. Principals (the who) are the entities in the system that can manipulate resources. Objects (the whom) are the resources in the system that require controlled access. An access-control model describes how access decisions are enforced in the system; the expression of a set of rules within the model is an access-control policy (the what).

Based on the analysis of the protection needs in web applications, it is clear that a hierarchical multi-level MAC model can address these needs. In such models, a system organizes its principals and objects into hierarchies based on their trustworthiness, and assigns appropriate privileges to each hierarchy. Access decisions are based on the hierarchy of the principals and objects. For example, SELinux and Windows Vista have adopted a MAC model to enforce restrictions on programs based on their trustworthiness.

We analyzed several existing multi-level MAC models such as Biba [7], Bell-LaPadula [6], and hierarchical protection rings (HPR) [33]. There are several similarities between the protection needs of web applications in web browsers and those of programs in operating systems. In operating systems, a program with user-level privileges must be isolated from a program with kernel-level privileges. In addition, the memory address spaces of user programs should be isolated from one another. Our design is primarily motivated by this similarity to protection needs in operating systems.

HPR was first introduced in the Multics operating system. In Multics, the access permissions are organized into hierarchical rings numbered from 0 to n (Figure 1). Ring 0 is the most privileged ring and ring n is the least privileged ring. The access permissions in a ring x are a subset of access permissions in ring y, whenever $x > y$. A process in a particular ring is limited to use the access permissions in its own ring or outer rings. There are special gates between rings to allow a process from an outer ring to request some resources from an inner ring in a controlled fashion. To isolate the memory address spaces of user programs, Multics uses segment descriptors. Organizing the program in rings provides separation of privilege, and memory isolation enforced via segment descriptors further increases the granularity of protection offered by

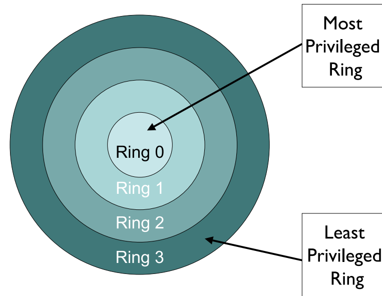

Figure 1: Protection rings: All principals and objects are organized into protection rings. The innermost ring is the most restricted ring and the outermost ring is the least restricted ring.

rings and enforces the principle of least privilege. ESCUDO is an adaptation of HPR to meet the protection requirements of web applications.

4 The ESCUDO Access Control Model

ESCUDO consists of four components:

- *Rings:* ESCUDO treats each web page as a "system," and all the elements in this system are placed in a static set of per-page protection rings. This is similar to HPR in operating systems. However, unlike in operating systems, where there is only one set of rings, a browser can simultaneously host multiple systems (i.e. web pages), the set of rings for each web page is independent from the others. The rings of web pages belonging to the same origin are compatible with each other.

- *Ring Assignments:* A web application should provide the ring assignments for all the elements in the system based on the trustworthiness of the elements and protection needs. The ring assignment method varies depending on the type of element and is discussed in section 4.1. This step is called "configuration," analogous to a system administrator configuring a system. Our configuration method provides fine-grained granularity in specification.

- *Access Control List (ACL):* ESCUDO allows objects to additionally use an ACL to improve the granularity of protection provided by the model. Essentially, the ACL used by each object enforces the principle of least privilege. Section 4.1 describes how an object can configure its ACL.

- *Access-control Model:* ESCUDO uses a MAC model based on HPR for enforcing access restrictions inside the browser. The access decisions are based on the configuration (ring assignment and ACL) provided by the application. The rules in the access-control model are described in section 4.2.

This design reflects the three principles we summarized in Section 2.2. With rings and ACLs, privileges in a web applications are divided into many pieces; these pieces are organized into a widely-used hierarchical model, making them easy to use. The fine granularity of principals and privileges is also achieved through the introduction of multiple rings. With fine-grained principals, fine-grained privileges, and well-isolated privileges, the principle of least privilege can be easily enforced in web applications.

4.1 Rings, Ring Assignment, and ACL

ESCUDO allows web developers to choose a set of rings for their applications, and assign the elements of the web applications into these rings. The set of rings for one web application is independent from that of

others; therefore, other than defining the relationships among different rings, ESCUDO does not define what each ring means, nor does it stipulate the total number of rings. The definitions are up to the web application designers. Designers can choose the total number of rings that fit their application needs; they can make their own ring assignment, independent of other applications.

Rings in ESCUDO are labeled $0, 1, \ldots, N$, where N is application dependent. In the HPR model, higher numbered rings have lesser privileges than lower numbered rings. Ring 0 is the highest-privileged ring, and ring N is the least-privileged ring. All examples in this paper, for the sake of simplicity in illustration, use $N = 3$. This is a large enough number to demonstrate interaction between rings without being cumbersome; other than that, 3 is arbitrary.

In this subsection, we describe how various principals and objects in the web application are assigned to rings. Web applications can communicate the ring assignment to ESCUDO either using HTML tags or optional HTTP headers, depending on the type of the object.

DOM Elements: Recall that DOM elements can act as both principals (e.g. script-invoking constructs) and objects (e.g. HTML excerpts). We use the HTML *div* tag to label each DOM element. HTML *div* tags were originally introduced to specify style information for a group of HTML tags; recently they have been extended for other purposes [35]. We introduce a new attribute called the *ring* attribute for the *div* tag. This attribute of the *div* tag assigns a ring label to all the DOM elements within the scope of the tag, which is the region enclosed by the *div* and */div* tags (Figure 2). We refer to such *div* tags as access-control (*AC*) tags.

```
<div ring=2 r=1 w=0 x=2>
  ...
  <div ring=3 r=2 w=0 x=2>
  ...
  </div>
</div>
```

Figure 2: Ring assignment

HTML allows hierarchical *div* scopes, i.e., a *div* scope can be enclosed entirely within another *div* scope. Therefore, ring assignments can also be hierarchical. To maintain the integrity of the ring assignment, ring numbers in the inner scope must be equal to or higher than the ring numbers in the outer scope (i.e. fewer privileges). Figure 2 gives an example of ring assignment. Special attention must be taken to ensure the integrity of the ring assignment. In Section 5, we will describe specific mechanisms to thwart attempts to compromise the integrity of ring assignment.

When a DOM element acts as an object, ESCUDO allows web applications to further specify a finer grained security policy on how this object can be accessed, in addition to the policy already imposed by the rings. ESCUDO uses Access Control Lists (ACL) for this purpose. Each ACL consists of three items: permissions for `read`, `write`, and `use` operations. The meanings for `read` and `write` operations are straightforward; the `use` operation needs more explanation. In some scenarios, web browsers implicitly access objects on behalf of principals, even though the principal does not explicitly request the access. For example, whenever an HTTP request is generated for a target URL, web browsers automatically attach the cookies belonging to the target site to the HTTP request. However, the principal who initiated the request did not explicitly reference the cookies. Another example is delivering a UI event to a DOM element using a JavaScript program. We call these implicit accesses the `use` operation.

An ACL is specified using a list of attributes (`r, w, x`) in the *div* tag, where `r`, `w`, `x` refer to the `read`, `write`, and `use` operations respectively. The value of each attribute identifies the outermost ring required for the operation. For example, in Figure 2, the outermost *AC* tag maps the objects inside its scope to ring 2 (“ring=2”). However, only principals in ring 0 can modify any DOM elements embedded inside

the outermost *AC* tags ("w=0").

Cookies: Typically, web applications instruct the web browser to store a cookie in the browser using a *set-cookie* header in HTTP. In ESCUDO, we use an additional optional HTTP header to communicate to the browsers the ring assignment and ACL for cookies. Cookies that contain sensitive data such as session identifiers should be mapped to a higher-privileged ring. Other cookies could be mapped to lesser-privileged rings. If ring mappings are omitted from the HTTP header, by default, all cookies are assigned to ring 0.

Native Code API: The ring mappings for native code APIs such as XMLHttpRequest are also communicated to ESCUDO using an optional HTTP header. By default, ESCUDO assigns native code API such as XMLHttpRequest to the highest-privileged ring 0, conforming to the fail-safe defaults guideline. Web applications may assign the native code APIs to different rings.

Browser State: ESCUDO mandatorily assigns internal browser state such as cache and browsing history to ring 0. In our current model, the ring assignment of browser state is not configurable. The web browser could manipulate or use the state information. However, JavaScript programs in the applications cannot manipulate the state, unless they belong to ring 0. This is because there are well-known attacks that abuse this information for tracking users [18].

4.2 The Mandatory Access Control Policy

ESCUDO defines a Mandatory Access Control (MAC) policy based on rings and ACLs, and this policy controls how principals in a web page can access the objects.

For the sake of presentation, we use the following notation for describing the policy: $\langle P \rhd O \rangle$ denotes a principal P trying to perform an operation $\rhd$ on object O. $\mathcal{R}(P)$ and $\mathcal{R}(O)$ denote the rings of the principal and object respectively. $\mathcal{O}(P)$ and $\mathcal{O}(O)$ denote the origin of the principal and object. We use $\sqcap(O, \rhd)$ to denote the least-privileged ring that is allowed to conduct the operation $\rhd$ on the object O. An access request $\langle P \rhd O \rangle$ is permitted if and only if the access is permitted by all the following three rules:

1. **The Origin Rule:** $\mathcal{O}(P) = \mathcal{O}(O)$

   Origin is the unique combination of $\langle protocol, domain, port \rangle$ in the URL of the web application that instantiates the principal or object. The origin rule requires the principal and object to belong to the same origin. However, unlike the SOP, this is not the only basis for access-control decisions.

2. **The Ring Rule:** $\mathcal{R}(P) \leq \mathcal{R}(O)$

   The ring rule factors the trustworthiness of the principals and objects into the model. The ring rule requires that the principal's ring should be of equal of greater privilege than the object's ring.

3. **The ACL Rule:** $\mathcal{R}(P) \leq \sqcap(O, \rhd)$

   The ACL rule further limits the access control on objects. The ACL rule requires that the principal's ring be at least as privileged as that specified for the operation by the object's ACL. Web applications can avoid interference between JavaScript programs belonging to the same ring by assigning a more restrictive ring in the ACL.

   However, it should be noted that web applications cannot associate an ACL with an object that is less restrictive than the object's ring. For example, an object assigned to ring n cannot have an ACL that permits a principal belonging to n', where $n' > n$, to access the object. Even if the ACL is set incorrectly, the ACL will be ineffective because the Ring Rule prevents such an access.

```
1   <html><head><title> Paul's Blog </title></head><body>
2   <div ring=2 r=0 w=0 x=0 nonce=23409750497590487 >
3     <h1>Scavenger Hunt!<h1>
4     <hr>
5     <h2>Paul: I will award the student bringing me the
       following items:</h2>
6     <ul>
7     </li>Yellow #2 pencil</li>
8     </li>Secretary's middle name</li>
9     </li>Number of ceiling tiles in our lab</li>
10    </ul>
11  </div nonce=23409750497590487>
12    <hr>
13    <h4>Comments</h4>
14    <div ring=3 r=1 w=1 x=1 nonce=23409750497590487>
15      Karthick: What will we get?
16      <div ring=0 r=0 w=0 x=0 ><script>
17         // malicious script that may modify the above list. //
18        </script></div>
19    </div nonce=23409750497590487>
20  <hr>
21  </body></html>
```

Figure 3: Assigning DOM elements to rings: This is the web page of a blog application. The original posted message is isolated from the user comments by assigning them to different rings.

4.3 An Example

To help understand our model, we give a more complete example in Figure 3. This is an example of a blog application. In Line 2, the original blog post (Lines 2-11) is assigned to ring 2 as a principal, and its ACL indicates that only ring 0 has the permission to `read/write/use` it [1]. The user comment (Lines 14-19) is assigned to ring 3, so even if there is a malicious script in the user comment, the script cannot access anything in the original blog post. If a ring specification is missing, ESCUDO assumes a safe default value, i.e. the ring attribute will be set to the least-privileged ring, and the ACL will be set to `r=0, w=0, x=0`, allowing only the principals in ring 0 to access it.

5 Security Analysis of Escudo

The key to Escudo's security enforcement is the safety and integrity of the configuration provided by the application. Because Escudo is a MAC model, Escudo reads the configuration information provided by the application and performs the ring mapping exactly once. Escudo's implementation disallows reassignment of rings, because the configuration information is not exposed to JavaScript programs for modification.

We describe additional measures to ensure the safety of the configuration from tampering. The configuration information for all the principals and objects maintained inside the browser is not exposed to JavaScript programs. However, because the ring mapping for DOM elements is communicated via HTML, it is vulnerable to certain tampering methods via HTML and JavaScript. Escudo enforces some additional rules to prohibit such tampering methods. Broadly, there are two ways that HTML or JavaScript could be used for illegally elevating privilege.

(1) **A Principal Increasing Privilege:** A JavaScript program may attempt to remap an *AC* tag to a higher privileged ring using the DOM API function *setAttribute*. Recall that the configuration information is not exposed to JavaScript programs. Therefore, such attempts to modify the attributes cannot succeed.

[1] Please temporarily ignore the number in the `nonce` attribute. We will explain the purpose of that attribute in Section 5.

(2) **A Principal Trying to Create a New Principal with Elevated Privilege:** HTML tags could be vulnerable to node-splitting because of vulnerabilities in the application [21]. In a node-splitting attack, an attacker may prematurely terminate a *div* region using *</div>*, and then start a new *div* region with a different set of ring assignments (potentially with higher privileges). This attack escapes the privilege restriction set on a *div* region by web developers. Node-splitting attacks can be prevented by using markup randomization techniques, which involve incorporating random nonces in the *div* tags (See Lines 2, 11, 14, and 19 in Figure 3). Escudo ignores any *</div>* tag whose random nonce does not match the number in its matching div tag. The random nonces are dynamically generated when constructing a web page, so adversaries cannot predict those numbers before they insert their malicious contents into a web page.

JavaScript programs can add new DOM elements. A malicious JavaScript program may attempt to use this feature to create a new *AC* tag with higher privileges. Escudo enforces a **scoping rule** to protect against such attempts. The scoping rule restricts all child elements of a DOM element to be mapped to either the same ring or some less privileged ring. Formally speaking, when a *div* tag is labeled with `ring="n"`, then the privileges of the principals within the scope of this *div* tag, including all sub scopes, are bounded by ring level n. Escudo's implementation strictly enforces this even if the ring specification of the sub scope violates this rule.

In a properly configured web application, a malicious principal would belong to the least privileged ring. As a result, such a malicious principal can only modify DOM elements that are mapped to the least privileged ring for write operation. That is, a malicious principal can add new DOM elements in only the least privileged ring. The scoping rule restricts all child elements of a DOM element to be mapped to either the same ring or a less privileged ring. As a result, a malicious principal cannot create a new principal that has higher privileges than itself. All the DOM modifications done using the DOM API are subject to the scoping rule.

6 Evaluation

We implemented a prototype of ESCUDO on the Lobo web browser and evaluated the prototype to ascertain the feasibility of deploying and using ESCUDO. Our evaluation assessed the following: (1) how web applications can take advantage of ESCUDO (2) compatibility with legacy web applications, (3) resistance to common XSS and CSRF attacks, and (4) performance overhead.

6.1 Implementation

We implemented a prototype of ESCUDO for the Lobo web browser [34], an extensible Java-based web browser. Lobo is intended to be a platform for building new client-side web languages. Therefore, the browser architecture is designed to be easily extensible. Implementing ESCUDO on Lobo involved 500 lines of code for extracting, tracking, and enforcing the ESCUDO policy specified by the web application. ESCUDO's implementation can be categorized into three parts: extracting the security contexts, tracking the security contexts, and enforcing the access control policy. The ESCUDO implementation maintains a security context derived from the configuration information provided by the application, tracks it through the browser, and makes it available whenever a principal makes a request. The security context is internally maintained data such as the ring assignments, domain, and ACL for all the principals and objects. We implemented the ESCUDO Reference Monitor (ERM), which enforces access-decisions based on the security contexts. The ERM is spread over several places because the places to embed the checks is specific to the object type.

6.2 Building ESCUDO-based Web Applications

We analyzed two open-source web applications, phpBB and PHP-Calendar, and created ESCUDO configurations for securing them. phpBB (http://www.phpbb.com/) is a multi-user message board application and PHP-Calendar (http://www.php-calendar.com/) is a multi-user online calendar application. We analyzed the principals and objects in these web applications and understood their security requirements. It did not take more than a day for modifying either application to use ESCUDO. A developer who knows the application better would be able to make the changes faster.

phpBB: phpBB is primarily used to create an online community, in which users may interact with one another by posting new topics for discussion, responding to existing discussion threads, or sending private messages to other users. The key security concern in phpBB is appropriately limiting the capabilities of messages posted by users. Table 2 describes the security requirements. Application contents, such as trusted JavaScript programs and HTML forms included into the web page by the application, require access to the messages, cookies, and the XMLHttpRequest object. Topics, replies, and private messages, however, do not require such privileges. Furthermore, user-provided topics, replies, and private messages are not expected to manipulate the contents of the web page. We created an ESCUDO configuration that enforces these requirements.

Principal	Modify Messages (DOM)	Access Cookies	Access XMLHttpRequest
Application contents	Yes	Yes	Yes
Topics and replies	No	No	No
Private messages	No	No	No

Table 2: Application contents can modify messages, access cookies, and access the XMLHttpRequest object. However, topics, replies, and private messages do not have such capabilities.

The ESCUDO policy for phpBB is described in Table 3. The head portion of the page contains style information and some trusted JavaScript programs. These are all assigned to ring 0 and can be manipulated only from ring 0. The content enclosed between the *body* and */body* tags is a mix of application provided content and user-provided topics, replies, and private messages. The body tags are assigned to ring 1 and can only be manipulated by principals in rings 0 and 1. Topics, replies, and private messages appearing inside the body are assigned to ring 3, but their ACL is configured so that they can be manipulated only by principals in ring 0, 1, and 2. Therefore, content provided by one user is completely isolated from content provided by another. There are two cookies in the web application, namely *phpbb2mysql_data* and *phpbb2mysql_sid*. Both cookies are assigned to ring 1. The cookies are attached only to HTTP requests generated by principals belonging to rings 0 and 1.

phpBB uses a template engine similar to *Smarty* for separating the HTML layout from the internal

Configuration	Cookies	XMLHttpRequest	Application contents	Topics& Replies	Private Messages
Ring	1	1	1	3	3
Access-control List					
Read access	≤ 1	≤ 1	≤ 1	≤ 2	≤ 2
Write access	≤ 1	≤ 1	≤ 1	≤ 2	≤ 2

Table 3: ESCUDO security configuration for phpBB: Application contents, cookies, and the XMLHttpRequest object are assigned to ring 1. The ACL for cookies and application-content is set so that it can be accessed only from rings 0 and 1. Topics, replies, and private messages are assigned to ring 3. The ACL for topics, replies, and messages are configured to allow only principals in ring 0-2 to manipulate it, providing isolation between the messages.

processing that produces content for the web page. To specify the ESCUDO configuration, we made changes in the template for each web page. Moreover, phpBB creates two session cookies and sends them to the browser using the *set-cookie* header. There were two places in the source code that create the cookies. We used the *header* function to add an additional HTTP header to specify the ring mapping for these cookies.

PHP-Calendar: PHP-Calendar is meant to facilitate a group's collaborative creating and tracking of events. An event in PHP-Calendar consists of a text message describing the event, time, and date of the event. The key security concern in PHP-Calendar is appropriately limiting the capabilities of events inside the web application. Table 4 describes the security requirements for PHP-Calendar. Application content requires privileges to modify events, session cookies, and use the XMLHttpRequest object. However, events should be prohibited from modifying other events via the DOM API and are not expected to manipulate cookies or use the XMLHttpRequest object. The security requirements for the PHP-Calendar application are very similar to phpBB.

Principal	Modify Messages (DOM)	Access Cookies	Access XMLHttpRequest
Application content	Yes	Yes	Yes
Calendar events	No	No	No

Table 4: Application content can modify messages, access cookies, and access the XMLHttpRequest object. However, calendar events do not have such capabilities.

Configuration	Cookies	XMLHttpRequest	Application content	Calendar events
Ring	1	1	1	3
Access-control List				
Read access	≤ 1	≤ 1	≤ 1	≤ 2
Write access	≤ 1	≤ 1	≤ 1	≤ 2

Table 5: ESCUDO security configuration for PHP-Calendar: Application content, cookies, and the XMLHttpRequest object are assigned to ring 1. The ACL for cookies and application-content is set so that it can be accessed only from rings 0 and 1. Calendar events are assigned to ring 3. The ACL for calendar events is configured to allow only principals in ring 0-2 to manipulate it, providing isolation between the events.

We created an ESCUDO configuration for enforcing the security requirements. Table 5 describes the ESCUDO policy for PHP-Calendar. In all the web pages inside PHP-Calendar, the body of the web page is a mix of application content and user created events. The content enclosed between the body tags is mapped to ring 1 and its ACL is configured to permit manipulation only by rings 0 and 1. However, as allowed by the scoping rule, the individual calendar events that appear within the body are assigned to ring 3 and configured to allow manipulation by rings 0, 1, and 2. Therefore, the various calendar events are isolated from one another. All the session cookies in the application are assigned to ring 1, along with the XMLHttpRequest object.

PHP-Calendar has created an HTML type system using PHP classes for separating the HTML layout from the internal processing required for producing content for the web page. This organization made it easier to modify the layout to incorporate the isolation policies. For specifying the ring mapping for cookies, we use the same technique as we used for phpBB.

Framework Support for ESCUDO Configuration: Creating ESCUDO configurations for static web pages is very straightforward because the configuration can be directly embedded in the web page and is not expected to change. In the case of web applications with significant portions of dynamic code, we need

more systematic methods for specifying the configurations. Otherwise, specifying the configuration will be cumbersome.

HTML template engines provide a structured method for isolating the view elements from the business logic. The view elements are specified in a template and data computed at run-time is plugged into the template to create the web page. The ESCUDO configuration can be specified in the template, isolating the configuration from dynamic data. Sophisticated template engines such as StringTemplate [29] provide a stricter separation between view and model, making it easy to manage ESCUDO configurations for large-scale web applications.

Language-based information flow could also be used to create ESCUDO configurations. The SIF framework is an extension of the Java Servlet framework to enforce confidentiality and integrity policies at run-time using language-based information flow [10]. In SIF, developer provides annotations in the source code to mark the confidentiality and integrity policies. These policies are then enforced at run-time when the program executes at the server. The confidentiality and integrity policies on the data can be used to automatically derive the ESCUDO configuration for the web page, when the web page is created. We are currently working on an SIF extension that could achieve this. We are unable to describe the extension in detail because of space limitations.

6.3 Compatibility with Legacy Applications

There are two types of compatibility concerns with respect to ESCUDO: (1) compatibility of ESCUDO-configured applications with non-ESCUDO browsers, and (2) compatibility of ESCUDO-based browsers with non-ESCUDO applications.

ESCUDO-configured applications are compatible with non-ESCUDO browsers. The only aspect that distinguishes an ESCUDO-based application is the availability of ring mappings for cookies, the XMLHttpRequest API, and DOM objects. For DOM objects, ring mappings are specified using *AC* tags, which are additional attributes in the `div` tag. Non-ESCUDO browsers would simply ignore these attributes. For cookies and the XMLHttpRequest API, ring mappings are specified using an optional HTTP header; they also will be ignored by non-ESCUDO browsers.

ESCUDO-based browsers are also compatible with non-ESCUDO applications. Non-ESCUDO applications do not provide any ring mapping. Therefore, all principals and object inside the application are assigned to a single ring, effectively mimicking the same-origin policy.

6.4 Defense Effectiveness

We evaluated the effectiveness of ESCUDO in addressing common XSS and CSRF problems. We created XSS and CSRF attacks for both applications. For the purpose of evaluation, we removed some protection mechanisms in the applications to facilitate the attacks. In both applications, we removed the input validation routines to facilitate XSS attacks. In phpBB, we removed the secret-token validation protection to facilitate CSRF attacks. PHP-Calendar had no protection mechanisms for CSRF attacks.

We created 4 XSS attacks for each web applications. In phpBB, we created XSS attacks for posting new messages on behalf of victim users and for modifying existing messages. In PHP-Calendar, we created XSS attacks for creating new events on behalf of victim users, and modifying existing events. All the attacks were neutralized in the presence of ESCUDO. This is because we structured the application to map all user-influenced regions to belong to ring 3.

We created five CSRF attacks for each web applications. We set up a malicious web site that crafted cross-origin requests for the two web applications, when accessed by a user. When accessed using our ESCUDO-enabled Lobo browser, the malicious site still issued the requests for the two web applications.

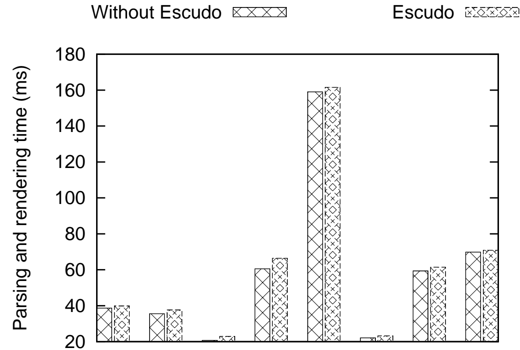

Figure 4: Performance overhead in parsing and rendering (in 8 different scenarios).

However, ESCUDO did not attach the session cookie automatically to the requests (because of the insufficient privileges of the principals), neutralizing the attacks.

6.5 Performance Overhead

ESCUDO's execution is invoked during both parsing and rendering of web pages and while responding to UI events. Therefore, to measure the performance overhead from using ESCUDO, we measured the slowdown in both activities. We instrumented Lobo to measure the amount of time taken to parse the web page and also to respond to UI events. In both cases, we did not observe any noticeable overhead in any of the activities. We setup 8 web pages varying amounts of *AC* tags and dynamic content. To measure the overhead we compared the time taken for parsing and rendering the 8 pages and averaged the rendering time over 90 executions (Figure 4). The average overhead was 5.09%. ESCUDO primarily does bookkeeping to keep track of the principals and this activity does not add any significant cost. Similarly, we did not notice any overhead for UI event handling.

7 Related Work

Same-origin policy (SOP) extensions: Jackson et al. [18] extends the SOP to browser cache content and visited link information to protect user privacy. Livshits and Ulfar [27] extends the SOP to additionally account for the principal names added to tag groups for neutralizing code-injection attacks. Karlof et al. [24] extends the SOP to account for certificate errors in the origin to distinguish resources in the authentic domain from a spoofed domain to detect dynamic-pharming attacks. While each of these proposals addresses a specific shortcoming in the SOP, they do not address the general gap between the fundamental model and the security requirements of modern web applications. ESCUDO is a fine-grained protection model specifically designed to meet the protection needs of modern web applications.

New browser architectures: The OP web browser isolates each web page instance and various browser components using OS processes [14]. The architecture makes communication between components explicit and interposes itself in all inter-process communication to provide isolation guarantees. Tahoma isolates each instance of a web application inside the browser using separate virtual machines [20]. The policy for identifying program boundaries and the permissible characteristics, such as which URL may be visited in each VM, are specified in a manifest. Essentially, these are two different approaches for isolating web applications from one another and limiting their permissible behavior. Both share the weakness that the granularity of protection is the web page, rather than objects within the page. In comparison, ESCUDO provides more fine-grained protection.

Chromium [5, 31] and Gazelle [38] are two new web browser architectures that bifurcate the browser into two portions, kernel and applications, for achieving better security and reliability. However, the access control mechanism is still based on the same-origin policy.

XSS and CSRF solutions: Current work has proposed several solutions for XSS and CSRF solutions. Approaches to XSS include taint-tracking [16, 28, 30], pure client-side solutions [26, 37], pure server-side approaches [9], and co-operating defenses [21]. Similarly, cross-site-request forgery solutions can be categorized into client-side methods [22], HTTP referrer header validation [25], proposals for new headers [4], and secret-token validation techniques [23]. All these solutions are attack-specific patches to the application, framework, or browser. In contrast to these solutions that address the symptoms of the underlying problem, ESCUDO is not a patch for XSS or CSRF problems. Rather, ESCUDO is a fine-grained protection model for web browsers. XSS and CSRF problems are thwarted as a side effect of addressing the fundamental weakness in the protection model.

In addition to patching, input validation and sanitization is a basic and primitive defensive coding technique for avoiding XSS. Frameworks such as PHP and ASP.NET provide libraries for this purpose. Filtering and sanitizing input, although useful as a sanity check, may be bypassed by known evasion techniques [15, 17]. As we showed earlier in the paper, ESCUDO prevents such attacks even when the front-line defense has been bypassed.

Mashup solutions: Mashups applications integrate content from several applications from differing origins into one web page. A key security concern in such applications is isolating the resources of each application from one another. Several frame-based design proposals for mashups have contributed new primitives and communication methods with minimal or no changes to the browser [3, 11, 13, 19]. Still, these proposals have a coarse-grained privileged model because they are based on the same-origin policy. Mashups are outside the scope of this paper. However, ESCUDO's fine-grained protection model could be extended to address security requirements for mashup applications by appropriately describing the relationship between the rings of applications from different origins.

JavaScript verifiers: There are several static and dynamic verifiers that could be used to verify conformance of a JavaScript program to a safe subset of the language [1, 2, 12, 39]. The primary target of these tools are applications that embed untrusted and semi-trusted JavaScript programs from third parties. Verifiers can be considered as an alternative approach to dealing with the web browser access-control failure. However, a publisher should trust the content provider to use the verifier on the JavaScript program. For example, a publisher may lease a portion of his page to an advertisement network. Currently, the publisher has to trust the advertising network to use a verifier on the JavaScript program provided to display the advertisement. In the case of ESCUDO, a publisher could take advantage of the browser protection model to enforce restrictions on the embedded JavaScript content rather than trusting an advertisement network. Furthermore, ESCUDO is generic protection model and constraints not only JavaScript programs, but also HTTP-request initiating principals. Therefore, ESCUDO can restrict the actions of an untrustworthy HTTP-request initiating principal manipulating more trustworthy resources (eg. CSRF attacks), but JavaScript verifiers cannot do this because these principals are outside the scope of their protection.

8 Conclusion

There is a disconnection between the protection needs of modern web applications and the prevailing protection model–same-origin policy. We outlined three characteristics that a protection model should have to address the disconnection. We presented ESCUDO, a new protection model that is systematically designed to fulfill the three requirements using mandatory access-control principles. We implemented a prototype of ESCUDO in the Lobo web browser, and illustrated how web applications can use ESCUDO to secure their resources using case studies. Our evaluations results indicate that ESCUDO is a practical access-control

model. In addition, ESCUDO can be incrementally deployed because it retains backward compatibility with legacy applications.

References

[1] Caja. http://code.google.com/p/google-caja/.

[2] Web Sandbox. http://websandbox.livelabs.com/.

[3] MashupOS: operating system abstractions for client mashups. In *HOTOS*, 2007.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *ACM CCS*, 2008.

[5] A. Barth, C. Jackson, and C. Reis. The security architecture of chromium browser. http://crypto.stanford.edu/websec/chromium/.

[6] D. E. Bell and L. J. La Padula. Secure Computer System: Unified Exposition and Multics Interpretation, 1976.

[7] K. J. Biba. Integrity Considerations for Secure Computer Systems, April 1977.

[8] M. A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] P. Bisht and V. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks . In *DIMVA*, 2008.

[10] S. Chong, K. Vikram, and A. C. Myers. Sif: enforcing confidentiality and integrity in web applications. In *USENIX-SS*, 2007.

[11] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *ACM CCS*, 2008.

[12] D. Crockford. ADSafe. http://www.adsafe.org.

[13] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *WWW*, 2008.

[14] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *IEEE S&P*, 2008.

[15] J. Grossman. Cross-site scripting worms and viruses. The impending threat and the best defense. http://www.whitehatsec.com/downloads/WHXSSThreats.pdf.

[16] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS*, 2009.

[17] R. Hansen. XSS cheat sheet. http://ha.ckers.org/xss.html.

[18] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW*, 2006.

[19] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW*, 2007.

[20] R. C. Jacob, R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *IEEE S&P*, 2006.

[21] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, 2007.

[22] M. Johns and J. Winter. RequestRodeo: Client-side protection against session riding. In *OWASP Europe*, 2006.

[23] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *IEEE S&P*, 2006.

[24] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *ACM CCS*, 2007.

[25] F. Kerschbaum. Simple cross-site attack prevention. In *SecureComm*, 2007.

[26] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *ACM SAC*, 2006.

[27] B. Livshits and U. Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *PLAS*, 2007.

[28] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[29] T. J. Parr. Enforcing strict model-view separation in template engines. In *WWW*, 2004.

[30] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, 2005.

[31] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys*, 2009.

[32] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[33] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3), 1972.

[34] J. Solorzano. The Lobo Project. `http://lobobrowser.org/`.

[35] M. Ter Louw, P. Bisht, and V. Venkatakrishnan. Analysis of hypertext isolation techniques for XSS prevention. In *W2SP*, 2008.

[36] A. Vance. Times web ads show security breach. `http://www.nytimes.com/2009/09/15/technology/internet/15adco.html`.

[37] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[38] H. Wang, C. Grier, A. Moshchuk, S. King, P. Choudury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *USENIX-SS*, 2009.

[39] K. Zyp. Secure Mashups with dojox.secure. `http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/`.