Least Privilege 2.0: Access Control for Web 2.0 applications

Sumeer Bhola Suresh Chari Michael Steiner
IBM T.J. Watson Research Center, USA
{sbhola, schari, msteiner}@us.ibm.com

2008-09-19

Abstract

Modern web sites make extensive use of scripting in the browser to provide a rich user experience. Further, these sites frequently put together content (including scripts) from different trust domains. Traditional models of access control which give the same level of privilege for all scripts on the browser are inadequate. Common web vulnerabilities such as cross-site scripting (XSS) and cross site request forgery (CSRF) can also be seen as failures of this principle of least privilege. Similarly, web delegation protocols, like OAuth, can be seen as attempts to introduce least privilege for cross-site access.

In this paper we propose a new access control model for the web. Specifically, we present a component model for web sites, where a site equates to a trust domain, and browser-side components to trust subdomains, such that components do not have all the privileges of the site. Additionally, our model expands the notion of access control for server accesses, to include browser to server, server to server, and intra-browser accesses, in a unified manner. This is essential to deal with intra-browser communication between trust (sub-)domains, and to extend web delegation beyond server to server accesses. We have developed new mechanisms and protocols to realize this model, and have implemented it in a web middle-ware platform, such that application writers deal only with high level abstractions. The implementation does not require any browser modifications.

1 Introduction

Web applications increasingly rely on extensive scripting on the client-side (browser) using readily available JavaScript libraries. These applications

typically contain content (including scripts) from more than one trust domain. In many cases, a site that includes content from different trust domains will endorse it such that it runs with the privileges of the site in the user's browser. Server-side access control is based only on identifying the user of the site, and does not know which content originated a request. Another example is websites containing display ads, where complete isolation of ad content is usually sufficient, and is typically accomplished using different browser frames (from different DNS domains) sharing the same browser window. In this case, the ad content has no user privileges to access the site's server.

To support more sophisticated applications, there has been much recent work [1, 2, 3, 4, 5, 6, 7, 8] to securely encapsulate content from different trust domains running in a browser window, as some form of *components*, while allowing the components to communicate. This *client-side communication* (between components) is done using a controlled interface: if content from two websites is encapsulated into separate components, B and C, then C can only affect B using the interface exposed by B to C. Our previous work on SMash [3], is a secure encapsulation model that was realized using browser frames.

However, these recent works have not adequately addressed the critical issue that valuable data, with confidentiality or integrity requirements, in web applications is eventually located on servers. Adopting web terminology, we will refer to such data as *resources* (we give a better definition in Section 3). In particular, the following issues arise when using client-side components.

1. Client-side communication bypasses server-side policy specification and enforcement: Sites write an access control policy for server-side resource access, that is enforced at the server. Client-side communication provides a way to bypass this enforcement, and hence access resources that one should not be permitted to. For instance, component B may have access to a resource at a site, and component C may be able to indirectly access that resource by communicating with B.

Current uses of client-side communication are often limited to basic user interface customization. However, access to resources using such communication has significant potential to decrease load at the server (by caching or computing at the client). In our opinion, writing a separate policy for client-side communication is too complex and error-prone and instead the same policy should be applicable everywhere. Additionally, the enforcement mechanism should handle policy changes, especially revocation of access

rights.

2. Same privileges for different components: A site may host multiple components, which, when run in a user's browser, may need or want to access their hosting site, or a different site. Giving the same rights/privileges to all components hosted by a site is inappropriate from a least privilege [9] perspective. Components may be written by different programmers, and subject to different levels of code validation. In fact, appropriate application of the principle of least privilege is very relevant to containing the impact of XSS (cross-site scripting) vulnerabilities, as we will elaborate on later. Typical deployments do exactly the opposite: all content provided by the site has the same privileges when running in the browser.

A site may also provide content from other sources, for instance ads written by others, or a site offering a proxy service, where the proxy service is proxying content from elsewhere. Such sites do a simple form of privilege separation using a handful of DNS domains, say two untrusted DNS domains, one for all ads and one for all content served through the proxy service. The remaining content may be served using a trusted DNS domain. The content from different DNS domains is served to different browser frames and is hence isolated: these could be considered different components. However, this simple approach has some drawbacks: (1) does not handle potential privilege escalation due to client-side communication, (2) coarse-grained privilege separation, since content is usually placed into two categories: trusted or untrusted, (3) poor usability, since the programming abstraction offered to a web application programmer is not high-level and includes low-level details on how security is implemented.

3. Narrow scope of cross-site delegation: Cross-site resource accesses, due to user's delegating rights from one site to another, are becoming more common on the web. In particular, web delegation protocols like OAuth [10], Yahoo's BBAuth [11], Google's AuthSub [12] are specifically designed to support the good security practice of limited delegation. Using OAuth-like terminology, these protocols deal with requests originating at a *consumer* site for resources at a *service provider* site.

However, these protocols have key limitations (see Section 6 for a detailed discussion). Firstly, the protocols are limited to direct accesses from the consumer to the servers of the service provider. They do not deal with indirect accesses, using client-side communication. For instance, a consumer component may communicate with a service provider component in a user's browser, and the latter communicates with its server. Secondly, the low-level protocols are designed assuming that requests originate from the consumer's

server, and not from a browser. Relaxing the second restriction would allow the consumer's server to have some of its components make direct accesses from the user's browser to the service provider's servers. This in turn leads to the consumer having to decide the privilege levels of the components it hosts, for accesses to other sites.

This paper addresses these deficiencies, by providing a new *unified* model for same-site and cross-site access control that covers various modes of accesses: server-server, client-server, client-client. The model equates a site and a trust domain, and allows a site to identify trust sub-domains, enabling it to apply the principle of least privilege to its hosted components. We develop a system design consisting of novel protocols and enforcement techniques to realize this model. In particular, our design considers usability for both the end-user and the web application programmer, and is implementable without requiring changes to browsers. We have implemented a prototype of the design as an extension to a web 2.0 middle-ware platform [13]. The abstractions offered by the middle-ware, both on the server and the client (in JavaScript), make it easy to use the model, without knowing any protocol or implementation details. Our approach also offers protection against common web vulnerabilities like XSS (cross-site scripting) and CSRF (cross-site request forgery).

The paper is organized as follows. In Section 2 we present that requirements that motivate our solution. Section 3 describes our access control model, and Section 4 the system design. Section 5 gives an overview of the implementation, and an evaluation. Section 6 discussed related work, and we conclude in Section 7.

2 Requirements

Here we discuss the requirements that motivate our solution. They are classified into four broad categories.

Code Identity: The solution should provide both *fine-grained* and *coarse-grained* identification of content/code making the request.

Fine-grained identity is needed to allow application of the principle of least privilege, and helps with XSS (cross-site scripting) vulnerabilities. XSS attacks occur when a site incorrectly combines content generated by itself with malicious content received in a request originating from a (non-malicious) user's browser. When the combined content executes in a user's browser, it typically has the full rights of the user for that site, since it has access to (or can exercise) browser-cached credentials, like cookies. With

fine-grained identity, all content returned by the site does not have the same rights, and the malicious content is limited to the rights granted to the combined content.

Coarse-grained identity is useful for encapsulation purposes, since exposing fine-grained identity may be meaningless in some cases - such as across sites.

Also note that some form of code identity is essential to guard against CSRF (cross-site request forgery). In a nutshell, CSRF attacks are possible when the server only validates only user credentials, e.g., cookies or basic authentication headers, and not the identity of the content making the request. This is a mistake since even cross-site requests result in submission of the user's credentials.

Policy: There are multiple policy requirements.
(1) Minimal changes: There should be minimal changes compared to how sites currently specify their resource access control policies. In particular, how access rights are granted to users should not need to change, since our focus is on executing content and the rights granted to such content. For instance, the approach should be neutral to whether the site administrator uses RBAC (role based access control) or not.
(2) Fine and coarse-grained delegation to content: We view the rights granted to executing content as some form of delegation of user's rights to that content. Analogous to the identity requirement, delegation could be either coarse-grained, say to a site, or fine-grained, say to a component hosted by a site. For coarse-grained delegation, e.g., as one might expected from end-user delegation decisions, the site receiving the delegation should be able to further delegate it in a fine-grained manner, say to its components, to follow the principle of least privilege.
(3) Unified policy model: The policy model should unify different modes of resource access: client-client, client-server, server-server.
(4) Policy changes: Policy changes at the server should impact all modes of accesses. In particular, decrease in rights (revocation) should be handled.

Usability: The solution should offer *high-level abstractions*, both declarative and programmatic, that make it easy to use the model without knowing about the implementation details. The solution should work on current web browsers.

Security: Clearly, the solution should be secure. However it is important to state our assumptions. We assume correctly implemented browsers, i.e., browser security mechanisms are not compromised. We assume correct functioning of the network infrastructure, i.e., DNS hijacking etc. are

not in scope. We recommend using HTTPS for client-server connections to mitigate most network attacks. The user is assumed to not make well-known security mistakes, like clicking through certificate errors (when using HTTPS). In addition, the user is assumed to be self-motivated - he will not subvert code (such as browser code, or content executing in the browser) in a way that will cause himself harm. The servers of a site are not compromised by malware and retain control of its private keys. Denial of service attacks and UI attacks (e.g., phishing) are not in scope.

Non-requirements A non-requirement is information flow: our focus is on access control involving two entities - a code entity acting on behalf of a user, that is requesting resources controlled by another entity.

The work is focused on client-side application of the principle of least privilege. We do not address componentization of code that is executing at the server of a site, i.e., code executing at server-side is monolithic and has all the rights of the site. There is a large body of previous work, both at the OS and language level, that can be applied to server-side componentization.

3 Model

This section describes our access control model. In Section 3.1 we define a web site and what it is composed of, including components. Then we describe the objects in an access control policy (Section 3.2), followed by how the subjects are identified (Section 3.3). Finally, we describe how access control is enforced in a unified manner for different modes of access (Section 3.4).

To make the description concrete, the model assumes that a browser frame is used to securely encapsulate a component. This is not essential – Appendix C discusses alternatives. The content of a component is represented using DHTML (Dynamic HTML, i.e., HTML, CSS and JavaScript). Without loss of generality, we adopt the SMash model [3] for the component interface: the interface consists of asynchronous communication *ports*. Once a component is loaded, this port interface is available to the content that loaded the component.

3.1 Site

We equate a (web) site to a trust domain, such that an administrator can decide what collection of resources comprise a site. A site has servers that are part of the trust domain. For convenience, we will refer to the *site*

server as representing a logical aggregate of the physical servers. This is a conceptual model - the same physical servers may be hosting multiple sites. In that case we assume adequate isolation between sites, say using some form of server virtualization.

3.1.1 Resources

A resource is some service provided by the site, for instance, reading Alice's email. It is more abstract than the use of resource in the term URI, although in an implementation there will be a mapping between the two. Additionally, the operation being performed, reading email in the above example, is part of the resource definition. The semantics of operations can be arbitrary, and may include reading, writing, creating or deleting data.

The sites resources can be accessed by resource *requests* received at the site. Each request attempts to access one (or more) resources. Resources can be *public* or *non-public*. Public resources can be accessed on behalf of an unauthenticated user. Non-public resources can only be accessed on behalf of authenticated users of the site, and are further classified as:

1. *Protected resource*: The code (of some site) making the request has to prove that it is acting on behalf of a particular user. This is typically implemented by the code proving possession of secret token(s) that represent the user and itself.

2. *Safe resource*: The code making the request can either prove that it is acting on behalf of a particular user, or prove that it is executing in a browser such that the user has authenticated using the browser. In the latter case, the code may not be in direct possession of tokens identifying the user. Safe resources may seem peculiar, but are needed to handle common web browser features: users entering URIs in the browser address bar, browser bookmarks (where there is no site-specific code making the request), and cross-site links in web pages. For instance, a user who has a long-lived authentication cookie for an email site, `email.com`, may enter the URI in his address bar and expects to see a page that displays his email inbox. Another way to view this category is that these are resources that need user authentication but do not need CSRF protection.

Resource requests can be done using HTTP or HTTPS. We require all non-public resources to be requested using HTTPS, otherwise the request will be denied.

3.1.2 Response Content

A response to a resource request contains *content*. We focus on content of type DHTML. For a request originating from a server, the content will be received by the requester. Before discussing requests originating from a browser, we give some background.

Background Content in a browser is represented in an object model called the Document Object Model. A browser window displays content from one or more browser *frames*, arranged in a hierarchy. A document loaded into a browser frame has a `location` property that represents the URI of the document. The `location` property of the *top* frame in the hierarchy is shown in the browser's address bar [1]. Browser frames can be dynamically created or deleted using JavaScript, and may be visible or invisible. The UI attributes of a frame, like visibility, size, can be dynamically changed.

For a request originating in a browser, the response can either be received by the requester, or it can be loaded in a separate frame in the browser. When the response is loaded in a separate frame, the notion of the site as a trust domain extends to that frame. For instance, when the user loads `email.com` in his browser window, it loads in the *top frame* of that window, and *belongs* to that trust domain. If this top frame were to include a script from `foo.com`, the executing script would also belong to the `email.com` site (trust domain). However, if it were to load content from `bar.com` in a separate frame, that content would belong to the `bar.com` site (even though it is a child frame of the top frame).

3.1.3 Components

A site contains/hosts a set of *component types*. Components can be requested using resource requests that are called component-load requests. These resources are always protected, and can only be requested from a browser. The response contains the content that will load and execute in the browser as a component. This content is always loaded in a separate frame.

Of all the frames that belong to a site, components are the only ones that are permitted to expose a service interface using client-side communication. This service interface can indirectly expose resources at the site server.

Figure 1 shows an example of a browser window containing four frames. The two components, belonging to `foo.com`, `email.com` are shown with a

[1] for uniformity of terminology, we are referring to what is called the top "`window`", as the top frame.

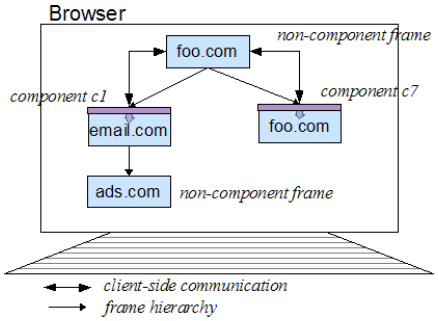

Figure 1: Browser-side Application

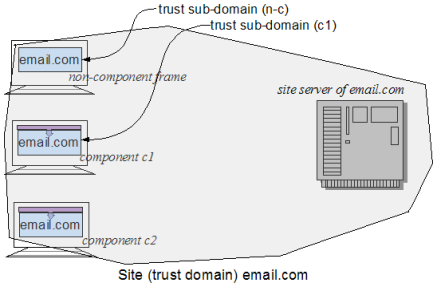

Figure 2: Trust Domain and Sub-domains

service interface that allows the top frame to invoke their service interface.

Each component type is considered a *trust sub-domain* of the site. The site can differentiate between component types, i.e., a component may not be allowed to act on behalf of a user for a particular resource request, while another component may be allowed. Trust sub-domains are not exposed to other sites, for encapsulation reasons, and because only the site hosting a component type knows how much trust should be placed in that component.

Additionally, non-component frames belonging to a site are all considered to be part of a trust sub-domain named `n-c`. Figure 2 depicts a site `email.com` with trust sub-domains corresponding to browser frames belonging to the site.

Discussion XSS attacks can be mitigated with this component abstraction. For instance, a site could serve all user-generated content using a particular component type, and assign no rights to that component type. Note that the end-user model of a site continues to be simple. An end-user only sees the URI of the top frame in a browser window, which is typically a non-component frame. The user does not need to know about the existence of components, or other frames in the hierarchy.

3.2 Objects in access control

Resources are the objects in access control. We are neutral to how the policy is specified: it could be either written by an administrator, or specified by end-users who own certain resources. In most cases, it will be a mixture of both combined with security (meta-)policy provided by the programmer as with J2EE [14] or Singularity [15]. To specify the policy, we adopt a simple *resource-name* abstraction for naming. A resource-name is a string name

for a collection of resources. An resource request is mapped to a resource-name, and the request is permitted only if it is permitted for each of the resource-names it maps to.

For usability, we allow resource-names to be parametrized by user. For instance, `read(x)` may represent read access to user `x`'s email, where `x` will be determined by the request. It is easy to extend this further, to allow other parameters, such as `read-department-docs(y)` where y is a department name. It can also be extended to allow hierarchical resource-names, which make is easier to express rights at different levels of granularity. However, for simplicity of exposition, we omit such extensions in this paper.

Example We consider a simple email site `email.com` with three resource-names, `read(x), write(x), config(x)`, which give read access (to all email of user `x`), write access (which includes the ability to send email as user `x`), read/write access to the user account configuration of user `x`.

The access control policy grants user `u1` the right to `read(u1), write(u1)` and `config(u1)`. User `u1` delegates some of his rights to site `foo.com`, because that site consolidates his email from different sites. Specifically, he delegates permissions `read(u1), write(u1)`. This delegation is also stored as a policy stored at the email site.

The email site hosts a component `c1`, that the administrator of the site trusts sufficiently to read and write email on behalf of its users. Therefore, this component is delegated rights `read(x), write(x)`. Non-component frames belonging to this site are only delegated the right `read(x)`. Finally, a component `c2` is allowed to change user configuration (including changing the delegation of rights to `foo.com`, and has rights `config(x)`. Note that these delegations to components are internal to the site, and decided by the administrator, and are not known by the users or other sites.

3.3 Subjects in access control

For requests to a site `s1`, we define the requester subject using a four tuple: `[u, s, c, r]`, where `u` is a user of site `s1`, `s` is a site making the request on behalf of user `u`, `c` is a component type hosted by `s` that gives finer-grained knowledge of which code in site `s` is making the request, and `r` is a set of resource-names (from `s1`) used to place an additional restriction on the rights available to `[u, s, c]`.

There is a special value of `c`, `ANY` which represents no finer-grained knowledge of the requester. When $s = s1$, it represents any browser frame (non-component or component) belonging to `s1`. As `ANY` is the union of all the trust sub-domains of `s1`, i.e., any component can also present itself as `ANY`, it

should not be assigned any rights which would be denied to some component trust sub-domain. When `s` $\neq$ `s1`, the value of `c` will always be `ANY` since `s1` is not expected to know about component-types of `s`. In this case, the request may be originating from the site server of `s`, or a frame belonging to `s`.

There is a special value of `r`, `ALL` which places no restriction on rights.

Example: same-site request Say the subject is `[u1, email.com, c1, ALL]`. Firstly, due to `u1`, the rights of the requester are limited to `read(u1), write(u1), config(u1)`. There is no further restriction due to `email.com`, since it is a same-site request. The rights of `c1` are restricted to `read(x), write(x)`, and this is applied by substituting `u1` as the value of `x`, and intersecting with the previously computed rights. This gives the final rights of `read(u1), write(u1)`.

Example: cross-site request The subject is `[u1, foo.com, ANY, {read(x)}]`. The rights are limited to those of `u1` intersected with the rights `u1` has delegated to `foo.com`, i.e., `read(u1), write(u1)`. Finally, the additional restriction {`read(x)`} is applied by using `u1` as the value of `x`, resulting in eventual rights of `read(u1)`.

Discussion The use of a resource-name set to restrict rights is mainly useful for cross-site requests: since site `s2` cannot identify its components to site `s1`, it needs a mechanism to restrict the rights its components have when making cross-site requests to `s1`. We could have chosen to drop the component type `c` from the `[u, s, c, r]` tuple, since `r` can provide similar functionality. However, including `c` provides a simpler mechanism (for same-site requests) since the subject does not encode the policy, making policy changes simpler.

This subject mechanism provides support for CSRF protection by identifying the site making the request. CSRF vulnerabilities occur when a site `s1` uses a subject mechanism limited to credentials expressing `[u, ANY, ANY, ALL]`, i.e., traditional user-only authentication via cookies or basic authentication, yet falsely assume subjects would correspond to `[u, s1, ANY, ALL]`, i.e., requests are limited to ones originating from its own site, and correspondingly and inadvertently assigns rights too broadly.

Fine-grained identification of a component, `c`, or restricting rights using the resource-name set, `r`, enable the application of the principle of least privilege. This provides an effective mechanism to contain the effect of XSS vulnerabilities - the frame that is exploited using an XSS attack is limited in its rights.

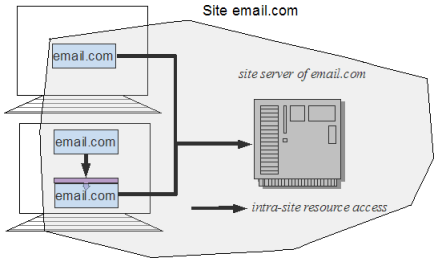

Figure 3: Same-site access

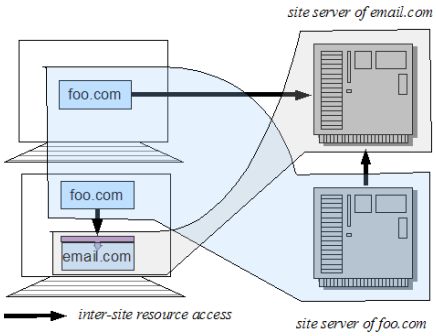

Figure 4: Cross-site access

3.4 Unified access control

The access control policy must be applied to the different modes of access: server-server, client-server, client-client. Figure 3 depicts the modes of same-site access: a component or non-component frame can either make a request to the site server, or make a request to a component in the same browser (using the component's service interface). Figure 4 depicts the modes of cross-site access from site `foo.com` to site `email.com`: the site servers communicating directly, a frame belonging to `foo.com` either communicating with the server of `email.com`, or a component belonging to `email.com`.

For requests arriving at the site server, we have previously outlined how the subject and the policy stored at the site server is used to decide the rights of the subject, and then whether the request should be permitted or denied. We now discuss how to extend this to client-side communication.

Recall that the service interface of a component is represented as a set of named ports, such that messages can be sent or received on each port.

3.4.1 Component port labeling

For each component type hosted by site `s1`, each port of the component is *labeled* with the set of resource-names that can potentially be accessed using that port and, hence, indicates the rights a caller on this port must have. This labeling is provided by the component writer, and validated by the site administrator, prior to hosting the component.

Example: Recall that site `email.com` hosts a component `c1` that displays emails, and allows the user to send emails. Say the component has two input ports named `ui` and `search`. The `ui` port allows customization of the user interface. The `search` port takes a search string as input, and

the component displays emails that match that search. These ports do not provide any direct access to the emails of the user. Hence, both ports apriori can be labeled with an empty resource-name set, {}. However, to prevent a malicious site from exhausting physical resources at site `email.com`, the site has defined an additional resource-name `search(x)`, that the `search` port is labeled with. Note, though, that port labeling only affects callers of a port: `c1` can provide a search feature through the UI, even in the case when the caller does not own the `search(x)` permission.

The site also hosts an invisible component (no UI, using a hidden frame) `c3` that has bi-directional `read, write` ports that offer read, write access to user email. By caching read email in the browser, `c3` reduces load at the site server. The ports are labeled {`read(x)`}, {`write(x)`}, respectively.

3.4.2 Load-time access control

The primary enforcement of access control is done when a component is loaded. The requester subject, `[u, s, c, r]`, is used to compute the rights of the requester in terms of resource-names. Then, each port label of the component to be loaded is examined, to check if the label is a subset of the requester rights. If it is, the port is enabled, else is disabled. The list of enabled ports is encoded in the component instance that is loaded and executed in the browser.

Example: cross-site load A subject `[u1, bar.com, ANY, ALL]` tries to load `c1`. Since user `u1` has not delegated any rights to `bar.com`, the component will be loaded with the `ui` port enabled and the `search` port disabled. A subject `[u1, foo.com, ANY, {read(x)}]` tries to load `c3`. Since `u1` has delegated reading and writing of email to `foo.com`, the requester has rights `read(u1)`, and the component will be loaded with its `read` port enabled and `write` port disabled.

Example: same-site load A subject `[u1, email.com, c2, ALL]` tries to load `c3`. Recall that `c2` had only been delegated rights `read(x), config(x)`, by the administrator. Therefore, `c2` will be loaded with the `write` port disabled.

Discussion In the above discussion, we have incrementally enabled ports in a component at load time. Another valid alternative would be to reject the request for loading a component unless all ports are enabled. The primary reason for our choice is that the extreme case, all ports disabled, reduces to the familiar security behavior on the web: a malicious site can typically load a (non-component) frame containing a user's email site, that shows his current email, since the frame does not do client-side communication.

In the above description, the policy decision of enabling ports is made at component load. We describe how to extend this to handle policy changes after load, in Appendix B.

4 System design

We describe the security aspects of a system design to realize the model described in the previous section.

4.1 Client-side isolation

To realize the model, we need to isolate content from different sites (trust domains), and additionally isolate trust sub-domains from the same site. This is done using browser frames.

Background A document loaded into a browser frame has a `domain` property which is the hostname of the server it was accessed from. The hostname is typically a DNS domain name (it can also be a numeric IP address). Browsers allow a frame to relax its domain [2]. For instance, code running in a frame with domain `c1.d.email.com` can relax it to `d.email.com` or `email.com`.

The browser *same origin* policy states that frames from different origins that are in the same frame hierarchy cannot examine or alter each others internal state. The origin includes the `domain` property and the protocol, for instance, HTTP or HTTPS. So a frame loaded from `email.com` using HTTP has a different origin than a page loaded from the same domain using HTTPS. Code running in a frame can read/write the internal state of all documents from the same origin.

A variant of the same-origin policy also applies to using XMLHttpRequest (XHR) to communicate with a server. Current browsers typically restrict a frame to perform XHR only to the hostname from which the frame was loaded. Cross-origin XHR (including Microsoft's version, XDR) [16, 17, 18], is becoming available in new browsers.

DNS domains for a site - loading content into a new frame Recall that a site server ensures that all non-public content (content that is the response to a non-public resource request) is accessed using HTTPS. Any attempt to access it using HTTP is denied. Also recall that content can be loaded from a site server into a new browser frame (non-component or

[2] The term domain, sub-domain when used alone will refer to DNS domain, sub-domain and not trust domain, sub-domain.

component). The site server uses multiple DNS domains, and controls the domain of the frame by controlling which content can be loaded from a particular domain. This control is necessary to ensure isolation between trust domains and between trust sub-domains. Additionally, the site server decides which DNS domains have the ability to act on behalf of the user, either as a component or non-component trust sub-domain.

Continuing the example of a site `email.com`, the rules are summarized as:

- Public content loaded into a new frame is from domain `email.com`. These frames cannot act on behalf of the user.
- Non-public (safe or protected) content that is not componentized is loaded into a new frame from domain `n-c.d.email.com`. This represents the trust sub-domain `n-c` that was described in the model. The corresponding subject for this frame is `[u1, email.com, n-c, ALL]`.
- A component type, say `c1`, is loaded into a new frame from domain `c1.d.email.com`. This frame represents subject `[u1, email.com, c1, ALL]`.

We mandate (but do not enforce) that well-behaved frames do not relax their DNS domains. This is in a frame's self-interest, since the only effect of relaxing its domain is to open itself to attack.

Loading content into existing frame Loading of content into an existing frame can be performed by the content already running in that frame. There are multiple ways to do this in browsers: same-origin or cross-origin XHR, image tags and script tags. We highlight the following issues related to security and usability:

- Cross-origin XHR is a recently announced browser feature, that is not available in most deployed browsers.
- Script tags, an alternative approach for cross-site requests, is generally deemed unsafe from a security perspective, since the included content is immediately executed. A consequence of this and the previous issue is that we have created a *usability* problem by using multiple DNS domains for a site. For instance, a frame in domain `c1.d.email.com` cannot safely load a public resource from `email.com`.
- The site server cannot necessarily distinguish between resource requests caused due to image or script tags in an existing frame, and

resource requests that will run in a new frame [3].

For these reasons, our system enforces the following rules:

1. *Proof of XHR*: Browser code using same-origin XHR requests must prove they are using XHR to the site server by including a custom request header in the HTTP request. Requests using script and image tags, or loading in a new frame, cannot include any custom headers. Cross-origin XHR specifications specify special headers that are automatically included by the browser, so the code making the request does not have to do anything additional to prove that it is XHR. Note that this proof only needs to be valid for requests originating from a browser - it does not matter that requests originating from a server pretend to be XHR (since the response is not going to a new browser frame).

2. *Usability*: Public resources can be retrieved from the domains reserved for safe and protected content, for example, `n-c.d.email.com`, `c1.d.email.com`, and using HTTPS, if the requester code in a browser is using XHR. Similarly, non-public content that is not componentized can be retrieved using XHR from the domains reserved for components, for example, `c1.d.email.com`.

3. *Non-XHR*: Resource requests using image or script tags must adhere to the domain constraints specified for loading in a new frame, as described earlier.

Client-side communication The client-side communication, and the associated ports are implemented using the SMash libraries [3]. Currently, this uses fragment identifiers as the low-level communication mechanism, but for new browsers the port abstraction can be implemented using the `postMessage` functionality in HTML5 [6].

4.2 Subjects for same-site requests to server

We describe how the system identifies the requester, when requests are made to the site server of `s`, and originate from a browser frame belonging to `s`. Our design is independent of how a user authenticates to a site server using

[3] The type of the response content, HTML or an image or JavaScript, could distinguish them. We do not consider that option due to the possibility of multiple content encodings for a response to a resource request, using the HTTP `Accept` header.

her browser. User authentication can be done using forms-based authentication, HTTP basic authentication, Microsoft's CardSpace [19], OpenID [**?**] etc.

- *User token*: When user authentication completes successfully, the user token for user `u`, represented as $t(u, s)$, is stored as a secure browser cookie[4]. The DNS domain of the cookie is a sub-domain of the one used for public content, so it cannot be accessed by frames that contain public content, but can be accessed by non-public content. In our example, it is stored as a secure cookie in the domain `.d.email.com`. This cookie can be explicitly read (and written) by frames in the domain `d.email.com` and its sub-domains, like `c1.d.email.com`, `n-c.d.email.com`. The lifetime of the token is decided by the site. We use the cookie mechanism as user re-authentication automatically refreshes the token for many frames, since the token is shared.

- *Component token*: The component token for a component `c`, represented as $t(u, s, c)$, is included in the component code at the time the component is loaded. Component loading will be described in Section 4.4. Client-side isolation ensures that the token cannot be stolen by other malicious frames, either belonging to the same site or a different site. Placing the token in the component, instead of storing it as a cookie at the time of user authentication, has a *scalability* benefit: a site may host hundreds of component types, only a few of which are loaded in any particular user's browser. With cookies, this would require setting hundreds of cookies.

- *Non-component token*: The non-component token for non-component frames belonging to site `s`, represented as $t(u, s, n - c)$, is stored as a secure browser cookie in the DNS domain of the non-component frames. In the example, this domain is `n-c.d.email.com`. The cookie is initialized when the user authenticates (see Section 4.4 for details). For the remainder of this section, we will ignore the difference between a component and non-component token, since conceptually they both represent a trust sub-domain of the same site.

Proof of token possession Requests may choose to prove possession of one or more tokens. Proving possession in a browser is proving the ability to read the value of a token. A request that proves possession of only token $t(u, s)$, is considered to be from requester `[u, s, ANY, ALL]`. A request that

[4]Secure cookies are sent by the browser only in HTTPS requests.

proves possession of *both* tokens $t(u, s)$ and $t(u, s, c)$ is from requester `[u, s, c, ALL]`. A request that does not prove possession of any token, but the request includes a cookie header with the $t(u, s)$ token is from requester `[u, ANY, ANY, ALL]`.

In our implementation, these tokens are generated in a straightforward manner by the site server using nonces and message authentication codes (MAC). The requester proves possession of a token by sending the token itself, since the tokens only flow over HTTPS channels where the receiver of the token has already been authenticated. An alternative would be to prove possession by performing client-side MACs.

4.3 Subjects for cross-site requests to server

We describe how the system identifies the requester, when requests are made to the site server of `s1`, and originate from either a browser frame or site server of another site, say `s2`. A cross-site subject of `[u, s2, ANY, r]`, where `u` is a user of `s1`, the component is ANY since `s2` does not reveal its trust sub-domains to `s1`, and `r` is a restriction (as defined in Section 3.3) is represented using a single *cross-site* token $t(u, s1, s2, r)$. This token could be generated in multiple ways: (1) by `s2`, using a shared secret decided previously between `s1` and `s2`, (2) by `s1`, using a protocol that allows `s2` to request such tokens. For (2), a protocol like OAuth [10] could be used [5].

Our implementation uses approach (1): `s2`'s site server generates the tokens, as `s2` does not trust its components or users browsers with the shared secret. For a component of `s2` to get such a token, it makes requests to `s2`'s site server, with its same-site subject. If the request is permitted, the response contains the token. The resources that return the token are marked protected, and the policy for access is subsumed by site `s2`'s access control policy.

Example Recall our earlier example where user `u1` of `email.com` had delegated rights `read(u1), write(u1)` to `foo.com`. Say `foo.com` has a component `c6`, that the administrator of `foo.com` has decided should only be able to exercise delegated rights `read(x)`. This component after being loaded in `u1`'s browser, makes a request for a `read(x)` cross-site token for `email.com`, to the site server of `foo.com`. The request includes same-site tokens $t(u1', foo.com)$, $t(u1', foo.com, c6)$, where `u1`' is the user

[5] Assuming protocol extensions for specifying `r`, and for allowing tokens to be used from browser-side code. The latter has been proposed as an extension - http://wiki.oauth.net/AccessorSecret.

of `foo.com` that is the same as user `u1` of `email.com` [6]. Since the access control policy permits `read(x)` tokens to `c6`, the response content contains token $t(u1, email.com, foo.com, \{read(x)\})$. Component `c6` can now make cross-site requests to the site server of `email.com` using this token. In our implementation, `c6` proves possession of this token by sending it in the request.

4.4 Loading a component

In our previous discussion on tokens, we have outlined how they are used to make resource requests to a site server. For component load requests, the protocol is more involved. First, we motivate the problem by describing the threats. Then we describe our secure loading solution in two stages, how user authentication is used to bootstrap a *secure loading* DNS sub-domain, and then how this used as part of the secure loading protocol.

4.4.1 Threats

There are two threats related to loading a component (A) theft of the component credential, (B) theft of the requester's credential.

Threat (A) occurs because the component token is contained in the response to the component load request. The site server is making the assumption that the response is going to be loaded in a new frame in the user's browser. However, this assumption may not be true. Consider the following scenarios:

- Cross-site attack: The site server of `foo.com` creates a token $t(u1, email.com, foo.com, \{read(x)\})$ and sends a request to the server of `email.com` to load component `c1`. The server responds with the component code containing the token $t(u1, email.com, c1)$. The `foo.com` site now has the component token of `c1`. If it is somehow able to forge or obtain the user token $t(u1, email.com)$, it can pretend to act on behalf of user `u1` as component `c1`.

- Same-site attack: Component `c2` tries to load component `c3`, where both components are from `email.com`. However, `c2` is malicious and would like to pretend to be `c3`. Since `c2` has possession of tokens $t(u1, email.com)$ and $t(u1, email.com, c2)$, it sends these tokens to a

[6] Site foo.com needs to know the mapping of `u1`' to `u1`, but `u1` could be a system-generated identifier by `email.com`, specifically generated for use by `foo.com`, and not the real-world identity, like `alice@email.com`. So this is not a privacy concern.

Browser code from frame `secureload.d.email.com` ⟷ Server `email.com`

⟶ **request**, URI: https://secureload.d.email.com/login, user: `u1` , password: secretstuff
⟵ **response**, set-cookie: load-t(u1, email.com), t(u1, email.com)
⟶ **request**, URI: https://n-c.d.email.com/inittoken, token: load-t(u1, email.com)
⟵ **redirect**, URI: https://n-c.d.email.com/inittoken, token: t(u1, email.com, n-c)
⟶ **request**, URI: https://n-c.d.email.com/inittoken, token: t(u1, email.com, n-c)
⟵ **response**, set-cookie: t(u1, email.com, n-c)

Figure 5: Token initialization at completion of user authentication

server that is colluding with `c2`. The server then makes the load request pretending to be `c2` and obtains the token $t(u1, email.com, c3)$. The server sends this token to `c2`, which can now pretend to be `c3`. This breaches the isolation between trust sub-domains.

Threat (B) occurs due to the manner in which the requester proves possession of tokens when loading a component. As a component is being loaded in a new frame, the proof of possession must be included in the request URI. Say the proof of possession is the tokens themselves. Once the component is loaded, it can read its own URI, which contains the tokens of the requester. Also, the component's URI is typically sent as part of the HTTP *referer* header, when it performs common actions like including scripts, images or traversing links. This can leak the tokens of the entity that loaded the component to an even wider audience. Note that even if proof of possession was limited to a MAC of the request URI (and not the actual tokens), the referer leakage may allow a malicious entity to perform a replay attack, i.e., load a component with certain ports enabled that allow it to escalate its rights.

4.4.2 User Authentication

We describe how user authentication is used to bootstrap a secure loading DNS sub-domain. In the context of our example, site `email.com` defines a DNS sub-domain `secureload.d.email.com`, such that user authentication is performed (or completed) using an authentication frame that has domain `secureload.d.email.com`.

Figure 5 shows the steps, when the site is using form-based authentication and user `u1` is providing his password. The first response message represents successful authentication, which triggers two tokens to be stored using set-cookie headers: (1) a *load token*, $load{-}t(u1, email.com)$ is stored as

a secure cookie in the domain `secureload.d.email.com`, and (2) a user token (defined in Section 4.2), $t(u1, email.com)$, is stored as a secure cookie in the domain `.d.email.com`. The next step is to initialize the non-component token (defined in Section 4.2) using a set-cookie header. To do so, the authentication frame creates a hidden frame with the URI shown in the next request message. The URI includes the token attribute as a URI parameter (we have chosen to separate it from the URI for clarity). In this case, the token is the load token set previously. To keep the load token a secret from all frames that are not in the `secureload.d.email.com`, the response is a redirect message that omits the load token and instead contains the non-component token as a URI parameter. The redirect causes an identical request to be submitted, and the response stores the non-component token as a secure cookie in the domain `n-c.d.email.com`. The hidden frame is then deleted.

4.4.3 Secure loading protocol

Our protocol redirects the load request through the secure loading DNS subdomain, to prove that the request is originating from the user's browser.

A load request is first submitted to `secureload.d.email.com`. The server validates the load token submitted as a cookie header, and additional tokens that prove the requester subject is `[u1, s, c, r]`. The submission of the load token as a cookie, coupled with the inability of any of the requesters to have possession of the load token, proves that the request is originating from the user's browser. Additionally, the server ensures that this is not a cross-origin XHR, by checking for special headers included by browsers. The only remaining possibilities are (1) request is for a new frame, (2) request is same-origin XHR, (3) request is for non-HTML content included using image and script tags. Option (2) is not possible since the requester must be a frame with domain `secureload.d.email.com`, i.e., the user authentication frame, which is trusted not to load component content using same-origin XHR. Option (3) is possible, but harmless, since the final HTML representing the component will be discarded by the browser which is expecting non-HTML content, and because component load requests do not cause any state changes at the server. So we are left with option (1), which is what we want. This addresses threat (A).

The server then creates a one-time token, $onetime{-}t(u1, email.com, s, c, r)$, and redirects the request to `c1.d.email.com` (say `c1` was being loaded), such that the one-time token is included in the redirected URI. It then validates the one-time token and decides whether to return the component content,

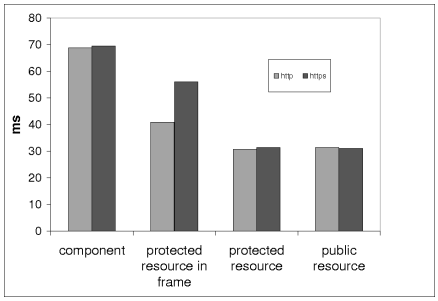

Figure 6: Same-site access latency

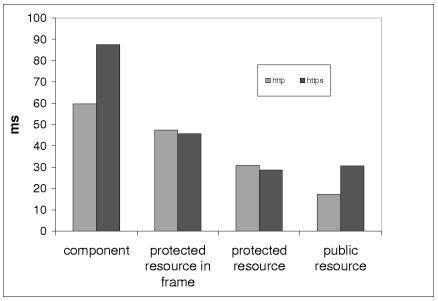

Figure 7: Cross-site access latency

with the embedded component credential for `c1`. Note that the one-time token is leaked to the component being loaded, and can subsequently be leaked to other sites due to the HTTP referer header. However, since the token has already been exercised before the component code is received by the browser, the token is of no value anymore. This addresses threat (B).

Appendix A illustrates the protocol with examples of same-site and cross-site component loading.

5 Implementation and Evaluation

We have implemented the design as a prototype that extends IBM's Project Zero web middleware platform [13]. Each server instance implements all functionality of a site. A server instance supports declarative configuration of components and their metadata, and declarative access control policies that encompass accesses permitted to users, components, and delegation to other sites. The details of the protocols and access control decisions are handled in the middleware. The application built on top of the middleware only has to deal with providing the DHTML for a component or resource once the request has been authorized. Additionally, we provide JavaScript libraries that client-side code can use to make resource and component-load requests, that automatically include the appropriate token. The inter-component communication is performed using the SMash [3] JavaScript library.

We have evaluated both the performance of our approach using micro-benchmarks and the usability by building applications.

Micro-benchmarks These measure and compare the performance of component and resource requests. Our metric is latency, since it is critical for interactivity of web applications. The latency was measured over multiple runs where each experiment run involved 100 iterations. The browser used

was Firefox 2.0, and all browser caching was turned off.

Figure 6 shows same-site access latency in milliseconds. Note that the difference in latency when using http versus https is typically negligible – this is due to the browser reusing an existing SSL/TLS connection. The difference in time between loading a component and a protected resource in a frame is attributable to the additional roundtrip of loading a component, and the creation and validation of a one-time token. The "protected resource" and "public resource" requests were performed using XMLHttpRequest – there is a negligible difference in latency between the two. Figure 7 shows cross-site access latency for similar tasks. One conclusion to draw from these experiments is that the additional overhead of the secure component loading protocol, compared to simply loading a protected resource in a frame, is minor.

The above experiments are for client-server access. Server-server access is expected to be similar for resource accesses, since it employs the same protocol and tokens. Note that component loading is not relevant for server-server access. Regarding client-client access, we have previously presented detailed results in [3], which employ a technique of communication using fragment identifiers. The event rate plot in that paper for Firefox 2.0 indicates a latency of 21ms, when a polling frequency of 10ms is used. This is acceptable for user interactivity. For new browsers that have native support for inter-frame communication (the postMessage interface in HTML5 [6]), client-client communication latency will decrease further.

Application One of the applications we have built is a used-car portal site which aggregates data from different dealers. The location of the selected dealer is shown in a map. There are four components in this application. One component is used to encapsulate the DHTML from each dealer. A second component encapsulates a Google map, and another component a list of dealers. Finally, a more trusted change-profile component has both read and write access to the user's profile. The map component never needs to see the user's profile, so it does not have read or write access to it. The dealer component has only read access to the profile. We found it quite easy to setup these rules declaratively.

6 Related Work

Since Saltzer and Schroeder's ground-breaking paper [9], significant work has been done in incorporating the principle of least privilege into distributed systems, applying it not only to rights of (human) users but covering also the

crucial aspect of (further restricted) rights of application code, e.g., in Abadi et al's logic of access control [20]. See Chapin et al. [21] for an overview of generic authorization frameworks.

However, little attention has been paid to least privilege in the world-wide web in the past, even though frequent attack vectors such as XSS or CSRF should make the benefit clear. A main obstacle was the lack of primitives which allow building web-application in composable fashion based on encapsulated and protected components. Increased security concerns with the popularity of Web 2.0, e.g., mashups of untrustworthy pieces such as ads, has recently led to a variety of work laying the foundation for such components, either based on current browser technologies [3, 1, 8], through language restriction and server-side analysis [4, 5], browser extensions [22] or standard extensions [6]. While our work is based on SMash [3], it is mostly independent of the underlying component framework. However, to handle also the most general case of mutually mistrusting trust domains or secure nested loading of sub-components, as is our goal, the framework should not have to rely on a trusted container environment provider. This in turn restricts the applicability of approaches such as Subspace [1] or ADsafe [4]. See also Appendix C for additional related discussion.

Many of these frameworks, though, leave the question of authorization open or cover only partial aspects, e.g., MashupOS [22] discusses client-side code principals but ignores user authentication and does not address the overall (distributed) systems aspect. In particular, none of them addresses authorization in a comprehensive and unified fashion covering client and server.

Largely independent of above component frameworks, a number of web delegation protocols have been developed in the last few years: First, proprietary BBAuth [11], AuthSub [12] and Windows Live [23] and, most recently, the community standard OAuth [10]. While these protocols have something in common with our approach, they share a number of deficiencies: Driven mostly by concerns related to delegation decisions by end-users, they don't encourage fine-grained rights restriction, e.g., they do not allow programmers and deployers to further sub-delegate and restrict rights. In fact, OAuth leaves out restriction of rights completely from the specification. Furthermore, neither of them target the case of client components communicating to client components or servers of a different trust-domain. With the increased popularity of component frameworks (see above) and cross-domain communication [16, 17, 18], this will become more important in the future. Additionally, our work moves beyond loading of components by a trusted entity (like gadgets loaded by Google), and allows for gadget

hierarchies, by providing a a secure component loading capability that can handle mutually distrusting entities. Lastly, unlike past work, we provide a unified and comprehensive framework for authorization and delegation decisions. Note that one could extend OAuth, which was developed only after most of our work was completed, e.g., to express the rights-restriction `r` in $[u, s, c, r]$ and to allow requests to originate from a browser. OAuth proxy [24] goes some way in that direction but is still more restricted than our approach.

Our work is also closely related to CSRF and XSS protection. While often not recognized as such, CSRF is in essence a code-identity problem and, ultimately, an issue of least privilege. While bolt-on approaches such as RequestRodeo [25] or cleaner browser models [26] do mitigate the effect of CSRF, only an authorization framework including explicit code principals will provide a clean solution to the underlying confused deputy problem [27]. Our approach does not directly prevent XSS attacks but can largely mitigate their impact. As comprehensive XSS prevention techniques, e.g., based on static analysis [28], are fairly involved and intrusive to deploy, most web sites resort to deploying Web Application Firewalls such as ModSecurity [29]. However, these are not easy to configure and are usually based on brittle heuristics. Therefore, our approach nicely complements these techniques by providing a second level of defense in limiting any remaining exposure.

7 Conclusion

We have proposed a new access control model for modern web sites that allows the principle of least privilege to be applied when specifying access control policies. This model includes a component model for web sites, such that components do not need to have all the privileges of the site. Our model expands the notion of access control, such that it handles browser to server, server to server, and intra-browser accesses in a unified manner.

We have presented new mechanisms and protocols that realize this model, and an implementation in a web middleware platform that does not require any browser modifications. The implementation exposes high level abstractions while hiding protocol details, making it easy to use, and has a low performance overhead. A side effect of our model is that it innately provides support for protecting against XSS and CSRF attacks.

References

[1] Collin Jackson and Helen Wang. Subspace: Secure cross-domain communication for web mashups. In *16th International Conference on the World-Wide Web*. ACM, ACM Press, 2007. (Cited on pages 2 and 24)

[2] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of HotOS XI: The 11th Workshop on Hot Topics in Operating Systems*. USENIX, May 2007. (Cited on page 2)

[3] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure cross-domain mashups on unmodified browsers. In *17th International Conference on the World-Wide Web*. ACM Press, 2008. (Cited on pages 2, 6, 16, 22, 23, and 24)

[4] D. Crockford. ADsafe. http://www.adsafe.org/. (Cited on pages 2, 24, and 32)

[5] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja — safe active content in sanitized Javascript. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf, June 2008. (Cited on pages 2, 24, and 32)

[6] Ian Hickson (Editor) and David Hyatt (Editor). HTML 5 — a vocabulary and associated APIs for HTML and XHTML. Technical report, W3C HTML Working Group, 2008. Working Draft available from http://www.w3.org/html/wg/html5/. (Cited on pages 2, 16, 23, and 24)

[7] Microsoft. Windows Live. http://home.live.com/. (Cited on page 2)

[8] Google. Gadget-to-gadget communication. http://www.google.com/apis/gadgets/pubsub.html. (Cited on pages 2 and 24)

[9] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975. (Cited on pages 3 and 23)

[10] OAuth core. http://oauth.net/core/1.0/, December 2007. Version 1.0. (Cited on pages 3, 18, and 24)

[11] Yahoo! Browser-based authentication (BBAuth). http://developer.yahoo.com/auth/. (Cited on pages 3 and 24)

[12] Google. Google account authentication (AuthSub). http://code.google.com/apis/accounts/AuthForWebApps.html. (Cited on pages 3 and 24)

[13] IBM. Project Zero. http://www.projectzero.org/. (Cited on pages 4 and 22)

[14] JavaTM Platform. Enterprise edition 5 specification. JSR 244, Sun Microsystems, 2006. (Cited on page 9)

[15] Ted Wobber, Aydan Yumerefendi, Martín Abadi, Andrew Birrell, and Daniel R. Simon. Authorizing applications in Singularity. In *Proceedings of the Second European Systems Conference (EuroSys2007)*, pages 355–368, Lisbon, Portugal, March 2007. ACM SIGOPS. (Cited on page 9)

[16] Anne van Kesteren (Editor). XMLHttpRequest level 2. W3C working draft, W3C, February 2008. (Cited on pages 14 and 24)

[17] Anne van Kesteren (Editor). Access control for cross-site requests. W3c working draft, W3C, September 2008. (Cited on pages 14 and 24)

[18] Sunava Dutta. Client-side cross-domain security. Whitepaper, Microsoft, June 2008. http://code.msdn.microsoft.com/xdsecuritywp. (Cited on pages 14 and 24)

[19] Microsoft. Windows CardSpace. http://cardspace.netfx3.com. (Cited on page 17)

[20] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993. (Cited on page 24)

[21] Peter C. Chapin, Christian Skalka, and X. Sean Wang. Authorization in trust management: Features and foundations. *ACM Computing Surveys*, 40(3), August 2008. (Cited on page 24)

[22] Helen J. Wang, Xiaofeng Fan, Collin Jackson, and Jon Howell. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 1–16. ACM, October 2007. (Cited on page 24)

[23] Microsoft. Understanding Windows Live delegated authentication. http://go.microsoft.com/fwlink/?LinkId=111425/. (Cited on page 24)

[24] Dirk Balfanz, Brian Eaton, and Eric Sachs. OAuth proxy. http://sites.google.com/site/oauthgoog/. (Cited on page 25)

[25] Martin Johns and Justus Winter. RequestRodeo: Client side protection against session riding? In *OWASP Europe Conference*, Leuven, Belgium, May 2006. (Cited on page 25)

[26] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, November 2008. ACM Press. (Cited on page 25)

[27] Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4):36–38, October 1988. (Cited on page 25)

[28] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2007)*, San Diego, CA, February 2007. Internet Society. (Cited on page 25)

[29] Breach Security. ModSecurity. http://www.modsecurity.org/. (Cited on page 25)

A Secure Loading - examples

Figure 8 shows an example of component `c6` from `foo.com` loading component `c1` from `email.com`. The first message is a request message to a URI in the `secureload.d.email.com` domain. The request also causes submission of a cookie containing the load token, even though the requester does not have possession of the load token. The submission of the load token proves that the request is originating from the user's browser, and not from a server. The response is an HTTP redirect to a URI in the DNS domain of component c1, `c1.d.email.com`, and includes a one-time token that encodes the requester subject. The redirect automatically causes the request to be resubmitted to the redirect URI. This results in the component `c1` being returned and loaded into a new frame in the `c1.d.email.com` domain.

Browser code from frame `c6.d.foo.com` ⟷ Server `email.com`

⟶ **request**, URI: https://secureload.d.email.com/c1, token: t(u1, email.com, foo.com, {read(x)}),
cookie: load-t(u1, email.com)
⟵ **redirect**, URI: https://c1.d.email.com, token: onetime-t(u1, email.com, foo.com, ANY, {read(x)
⟶ **request**, URI: https://c1.d.email.com, token: onetime-t(u1, email.com, foo.com, ANY, {read(x)}
⟵ **response**, Component `c1`

Figure 8: Secure cross-site loading

Browser code from frame `c2.d.email.com` ⟷ Server `email.com`

⟶ **request**, URI: https://secureload.d.email.com/c3, tokens: t(u1, email.com), t(u1, email.com, c2),
cookie: load-t(u1, email.com)
⟵ **redirect**, URI: https://c3.d.email.com, token: onetime-t(u1, email.com, email.com, c2, ALL)
⟶ **request**, URI: https://c3.d.email.com, token: onetime-t(u1, email.com, email.com, c2, ALL)
⟵ **response**, Component `c2`

Figure 9: Secure same-site loading

Figure 9 shows another example, of component `c2` from `email.com` loading component `c3` from the same site.

Note that in the both examples the user token for site `email.com` is also submitted as a cookie in all requests. We have omitted it from the protocol messages since the server ignores it.

B Policy changes for client-side access control

Section 3.4 had described client-side access control assuming the policy did not change. Here we discuss how to handle policy changes. To begin, we elaborate on server-side access control, to point out some subtleties in cross-site access control. Subsequently, we describe our approach to client-side access control.

Server-side access control is usually straightforward from a policy change perspective. When the site server of `s1` receives a request with subject `[u, s, c, r]`, there are two cases to consider:

- `s` = `s1`: In this case, the current rights of component `c` (which could be a specific component, or `ANY`) are stored at `s1`'s server and so the latest policy can be applied. The value of `r` is typically `ALL` in this case.

- $\texttt{s} \neq \texttt{s1}, \texttt{s} = \texttt{s2}$: In this case, the current rights delegated by `u` to site `s2` are also stored at `s1`'s server and the latest policy can be applied.

In the second case, `s1`'s policy change requirements are taken care of. However, `s2`'s requirements are not. The request could be originating from client-side code that is part of `s2`'s trust domain. The `s2` server has made a decision that this client-side code can exercise a subset of the rights delegated by `u` to `s2`: this subset (this could be the complete set, `ALL`), is encoded in `r` (recall that $\texttt{c} = \texttt{ANY}$ for cross-site accesses). If the server of `s2` changes its policy, the `s1` server does not know about it.

The solution to this issue is simple. Recall that the system design (Section 4.3) encoded this subject as a cross-site token $t(u, s1, s2, r)$. Two existing techniques can be used to handle policy changes by `s2`: (1) give the token a limited lifetime, (2) include a globally unique identifier in the token such that `s2` can tell `s1` to revoke tokens with certain identifiers. The latter approach leads to faster revocation. We represent this explicitly as a token $t(u, s1, s2, r, gid, t)$, where gid is the unique identifier, and t is the *expiry time*.

Next, we use this background to construct an example that illustrates the problem with policy changes for client-side access control.

Example In user `u1`'s browser, a component from `s2`, represented as subject `[u1, s2, ANY, r1]`, loads component `c1` from `s1`, represented as `[u1, s1, c1, ALL]`, which then loads component `c2` from `s1`, represented as `[u1, s1, c2, ALL]`. Component `c1` enables some ports for the component from `s2` and component `c2` enables some ports for component `c1`. We consider three kinds of policy changes: **(1)** the rights of `c1` are decreased by site `s1`, such that some of the enabled ports of `c2` give `c1` more rights than it should have, **(2)** user `u1` decreases the delegated rights to site `s2`, such that some of the enabled ports of `c1` include rights that are no longer delegated, **(3)** site `s2` decreases the rights of its component to act on behalf of `u1`, but the enabled ports of `c1` allow it to escalate its rights.

B.1 Solution

We use the approach of *port expiry* and *port refresh* to solve the problem described in the previous example. Any enabled port of a component that has a non-empty label, i.e., it gives access to some resources, has an expiry time associated with it. After the expiry time is reached, the port will be disabled. The port refresh mechanism is used in some cases to extend the expiry time - it will typically be used before expiry. The code that loaded

a component is informed of state changes (enable/disable) to ports using a control port that is always enabled.

Port expiry The port expiry time is decided using two values, a *soft expiry* time and a *hard expiry* time. The former is decided using a *freshness* interval configured by the administrator of the site hosting the component, say 1 hour. It is meant to ensure a certain degree of freshness of the policy being enforced by the ports. The latter is relevant for cross-site component loads when the requester had a token that contained an expiry time. The next expiry time is the minimum of the soft and hard expiry time. If the next expiry time is due to soft expiry, a port refresh will be initiated prior to expiry. If the expiry is hard, the port will be disabled and never reenabled. We are assuming rough clock rate synchronization between the browser and the server.

Port refresh To do a port refresh, the component needs to identify itself and the code that loaded this component. The former is done using the component token. For the latter, when the component is first loaded, it is also given a *port-refresh* token representing the subject that loaded it. This token does not allow the component to impersonate the subject in the token, and is limited to performing port refreshes. Given these two tokens, the port refresh protocol is simple - the component sends the tokens to its site server, which responds with a list of ports that should be enabled, and the next soft expiry time.

This simple approach to port refresh may impose significant load on the server. Note that port refresh is a pull protocol - pushing port updates from the server to the components is typically not an option for multiple reasons: (1) browsers permit a limited number of long-lived open connections to a server, (2) scalability problems at the server in the presence of a long-lived connection with each component instance. We adopt a *lazy refresh* optimization: a component which is idle, i.e, it is not communicating on ports that have non-empty labels, does not need to refresh even if soft expiry has occured. The refresh is delayed till the next time it becomes non-idle.

Example Revisited We revisit the previous example of policy changes, now using the mechanism of port expiry and refresh.

(1) Due to soft expiry, component `c2` sends a port refresh request to the site server of `s1`. This contains its token, $t(u1, s1, c2)$ and the token identifying its parent component, $\mathit{refresh} - t(u1, s1, c1)$. Due to a revocation of the rights of component `c1`, the server responds that all ports except for the control port should be disabled. The component enacts this decision and informs `c1` about the change in state of ports, using the control port.

(2) Due to soft expiry, component `c1` sends a port refresh request to the site server of `s1`. This contains its token, $t(u1, s1, c1)$ and the token identifying its parent component, $\mathit{refresh} - t(u1, s1, s2, r, \mathit{gid}1, t1)$. Due to a revocation of delegated rights by `u1` to `s2`, the server responds that all ports except for the control port should be disabled. The component enacts this decision and informs `s2`'s component about the change in state of ports.

(3) There are two ways site `s2` can revoke the rights of its component to access site `s1`: by informing `s1`'s site server of the revocation, or by relying on expiry of its component's cross-site token. We consider each in turn:

- Soft Expiry: Due to soft expiry, component `c1` sends a port refresh request to the site server of `s1`, containing its token $t(u1, s1, c2)$, and the token representing its parent $\mathit{refresh} - t(u1, s1, s2, r, \mathit{gid}1, t1)$. The server of `s1` notices that `s2` has revoked the token with identifier $\mathit{gid}1$, and responds that all ports (except the control port) should be disabled.

- Hard Expiry: Component `c1` observes that the expiry time $t1$ in the token of its parent $(\mathit{refresh} - t(u1, s1, s2, r, \mathit{gid}1, t1))$ has been reached. It disables all ports with non-empty labels and informs its parent. The parent then obtains a new cross-site token from its site server (`s2`), and uses it to reload a new instance of component `c1` to replace the old instance. The new instance has a hard expiry time corresponding to the new cross-site token.

C Non-frame alternative for secure encapsulation of a component

In Section 3, we assumed using browser frames to securely encapsulate a component. An alternative is to encapsulate components using an object capability subset of JavaScript, such as ADsafe [4] or Caja [5]. In an object-capability language, an object can only cause effects outside itself by using the references it holds to other objects[7].

This approach allows placing of multiple components in the same frame. First, some *container code* would load into the frame. The container would then load the individual components which are written in an object capability language. Note, that an object-capability language protects the outside world from the object, and not the object from the outside world. Therefore,

[7] Definition taken from the Caja specification.

the container code, and the server it is loaded from, needs to be completely trusted by the components. When considering two different sites (different trust domains), it may be difficult to find a third-party site that is completely trusted by both and can provide the container. Therefore, we continue to use browser frames to securely encapsulate content from different sites. However, within a site, the different trust sub-domains must already trust the site server. Therefore it is acceptable to extend this trust to container code provided by the site server. This allows multiple components from the same site to be in the same browser frame.

Tokens Each container is completely trusted by the site server and can completely act on behalf of the user. It uses these rights to get lesser privilege component-tokens for the components it loads and gives them to the components it loads. A component can use its component-token to make same-site resource requests, or ask the container to exchange it for cross-site tokens for cross-site requests.

Same-site component loading Component loading within a site does not cause creation of a new frame. The container code loads the new component's code, and uses the requester's component-token to ask the server about which ports should be enabled. It then instantiates the new component with the appropriate ports enabled.

Same-site component loading As before, component loading across sites causes creation of a new frame. Specifically, the container of the requesting component loads a container of the site from which the new component is being loaded. The new container is given the requester's cross-site token. It then loads the new component and decides which ports to enable in the same manner as with the same-site requests.