A Resource Management Approach to Web Browser Security

Jun Li
University of Oregon
Carlos III University of Madrid
Institute IMDEA Networks
Email: lijun@cs.uoregon.edu

Dongting Yu, Luke Maurer
University of Oregon
Email: dongtingyu@gmail.com, maurerl@cs.uoregon.edu

***Abstract*—While today's web browsers support multiple principals (i.e., web frames with embedded JavaScript code, or plug-ins) from many different origins at the same time, they do not have a clear resource management model, and the loose control on resource access has led to various types of web-based attacks. In this paper, we present a resource management framework for web browsers that allows both users of a web browser and the owner of a web page to specify their resource access control policies—which are then enforced by the framework's resource reference monitor. With our resource management framework, a web browser can become more secure, and we show that popular web attacks such as frame hijacking, cross-site request forgery, and DNS rebinding attacks, can all be addressed easily by deploying correct security policies. We also discuss how our resource management approach may be deployed and what a new paradigm it can bring to counter web-based attacks.**

***Index Terms*—web browser security; web resource access control; web reference monitor; web security**

I. Introduction

Traditionally, web documents are entirely static content, and a web browser is a client for downloading and rendering such content from remote web sites. Two features later developed, however, complicated matters. One is that a web browser can concurrently load documents from different origins, each into a different *frame*. The other feature is that a web document can include program code written in JavaScript to be interpreted by the browser, effectively converting the web document into an *executable*. As a combined result of these two features, inside a web browser there can be multiple *principals*, each principal corresponding to either a frame loaded with an executable web document or a plug-in.

This new paradigm makes a web browser a common environment shared by principals from all different origins. A principal not only has access to resources of the browser and its own, but also the resources of other principals. As a result, a malicious principal (e.g., www.attacker.com) could launch various attacks toward other principals or the browser itself. For example, a malicious frame can cause another frame to navigate a phishing site instead of a legitimate site. A malicious gadget in a mashup site (e.g., *iGoogle*) can replace another benign gadget with a spoofed one. Or, in a cross-site request forgery (CSRF) attack, a malicious web site www.malicious.com can invoke the browser to send a request to another web site www.honest.com in the name of the user, effectively impersonating the user by "hijacking" the browser.

Clearly, it is critical that all principals be protected from one another. The key to achieving the protection is two-sided: (1) Different principals must be *isolated* from each other, each with its own resources; and (2) the web browser must support the classical *reference monitor* concept, and access policies must be defined and enforced so that access to resources can be controlled systematically.

For isolating principals, one of the most important security policies in place today is the Same-Origin Policy (SOP): Two web documents or scripts can access each other if and only if they are from the same origin, where an origin is defined by a tuple `<protocol, host, port>`. The Google Chrome browser [1], and research browsers such as Gazelle [2], Tahoma [3], and the OP browser [4], have further proposed new browser architectures to support multiple principals and allow each principal to have its own protection domain.

Related to the "reference monitor" concept, what have been studied or deployed are mostly security policies that specify how principals may interact with each other under certain circumstances. For example, in determining which other frames a frame can navigate, one such policy is the descendant policy: A frame can navigate only its descendants [5], i.e., frames contained in it at any depth. Or, in determining if a web page from one server can include contents from another server, a mutual-approval policy can be adopted that requires both web servers to endorse each other for inclusion and being included [6]. These are worthwhile policies, but they are *ad hoc* solutions to particular problems.

In contrast, a modern operating system is able to separate processes cleanly and ensure every process executes within its own logical space. A process can only access a system resource if the resource is explicitly made available to the process. This allows processes to be protected from each other.

A principal in a web browser can be treated similarly to a process except that a process is loaded from hard disk by an operating system while a principal is loaded by a web browser from a remote server. Just as an operating system is a resource allocator for processes, a web browser should also be viewed as a resource allocator for its principals.

Few studies have systematically looked at how a browser

should define, allocate, and manage resources for principals, and how a resource management approach may improve the security of a browser. In this paper, we propose a resource management approach for a browser to protect its principals from each other while supporting inter-principal interaction and communication. Our contributions are mainly to:

- Identify resources typically associated with a principal;
- Define a resource management framework in browsers;
- Outline a language for describing resource policies;
- Illustrate how proper policies can secure a principal from various vulnerabilities; and
- Promote a new approach to addressing web-based attacks by installing correct security policies, as opposed to patching web browsers and web sites.

Our discussion does not target a specific web browser; it can apply either to new browsers or to upgrades of existing ones.

The rest of the paper is organized as follows: We first describe our resource management approach to web browser security in Section II. We sketch the semantics of the language specifying access control policies in Section III. We then exemplify how the proposed approach can address several well-known web security problems in Section IV. In Section V we discuss important issues related to our resource management approach, and we conclude our paper in Section VI.

II. Design

We propose in this section our resource management approach to protecting principals from each other and improving web browser security. We propose a **resource management framework** for a web browser. In this framework (Fig. 1), the resource allocation process must be coupled with security policy enforcement: whenever a principal is obtaining access to a resource, a **resource reference monitor** will verify the access against both mandatory and discretionary resource access control policies. The policies are specified using a language whose semantics we outline in Section III.

A. Resource Allocation Model in Web Browser

A principal in the web browser can be either a frame or plugin content: both can perform actions and interact with resources. A frame can include JavaScript, display rendered HTML code, or include other frames as well as plugin content. Plugin content runs in a user-installed browser plugin with the privileges of the plugin.

Every principal in a web browser has its own logical space, including its resources. Resource allocation happens when a principal is created and instantiated:

- When the browser first loads a page, it loads the page into a top-level frame, displaying it and processing any JavaScript code it contains. This frame is a principal in the browser, which will allocate certain resources to this principal.
- The top-level frame can further specify child frames or plugin components using HTML and JavaScript code. When invoked, each of these child frames or plugin components is also a principal, and a child frame can have its own child frames or plugin components recursively. In doing so, the child principal can inherit certain resources from the parent principal (e.g., a subset of the display area the parent uses) and obtain new resources from the browser.

Fig. 2 shows two principals and their creation and resource allocation in a web browser.

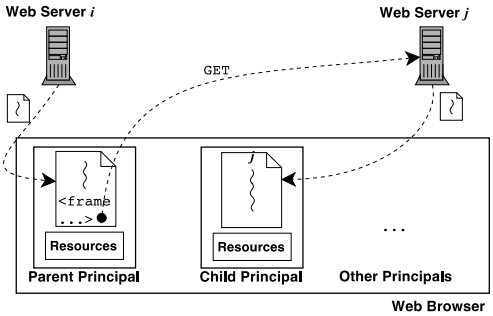

Fig. 2. Resource allocation example in a web browser. The web browser loads a web page from a web server i, creating a (parent) principal together with its resources. This principal then requests another web page from a web server j, creating a child principal together with its resources.

B. Global and Local Resources

Among the resources that a principal may access, some are global in that they are not bound to a specific principal; some are local as they are tied to specific principals. Below we list typical global and local resources. Our resource management framework can easily add new types of resources when needed.

Typical global resources include:

- *File system*: The file system of the computer that the web browser is running on.
- *Display*: The display associated with the browser window.
- *Inbound network channel*: The network channel for receiving inbound data or commands.
- *Outbound network channel*: The network channel for sending outbound data or commands.
- *JavaScript engine*: The engine that is responsible for interpreting and running JavaScript code. If a principal does not have access to the JavaScript engine, all its JavaScript code will be ignored.
- *Plugins*: Similar to the JavaScript engine, a plugin can interpret plugin content from a web page. Examples of plugins in current web browsers include the Adobe Flash add-on, the Java virtual machine, and QuickTime.

Typical local resources include:

- *Display area*: A specific area from the browser display that is allocated to a principal to use.
- *Inbound network channel from a specific site(s)*: A principal's network channel for receiving inbound data or commands from a specific site(s).
- *Network outgoing channel to a specific site(s)*: A principal's network channel for sending outbound data or commands to a specific site(s).
- *Cookies associated with a specific site*: Local data (usually name-value pairs) that record a user's identity, status, or other information of a web site.
- *Principals*: A parent principal can treat its child principals as a local resource.

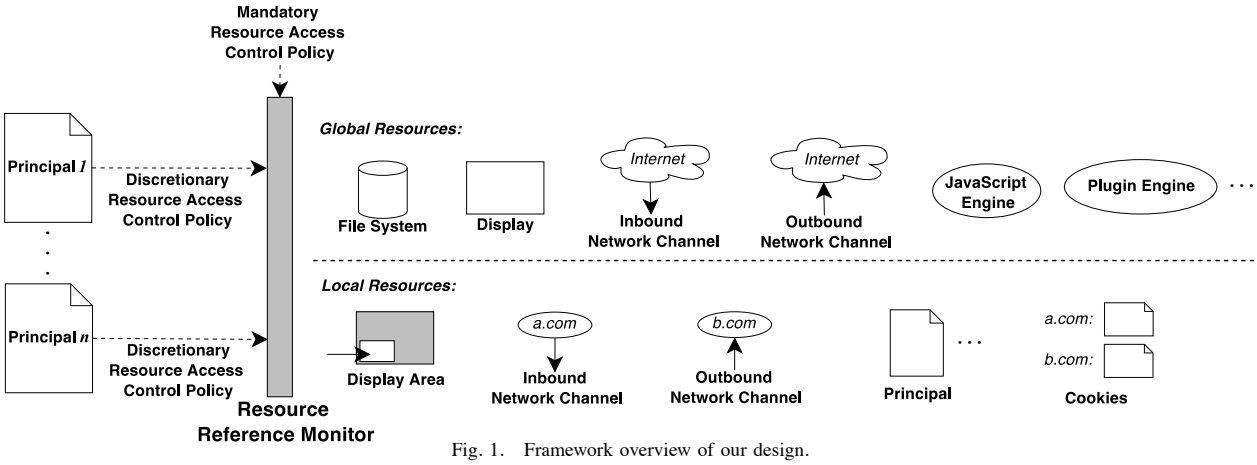

Fig. 1. Framework overview of our design.

C. Mandatory and Discretionary Resource Access Control

If a principal can access a resource, such access must be permitted by *both* the mandatory resource access control policy at the web browser level and the discretionary resource access control policy specified by relevant principals.

The mandatory control applies to all principals and cannot be modified by any individual principal. A browser's mandatory resource access control policy comes from the specification and configuration of the user. First, the user of the web browser can specify whether certain principals can have access to certain resources; and if so, what type of access rights a principal may have. The user can also specify the default access rights for various principals on different resources. Second, the user of a web browser can also configure its preferences related to resource management. The configuration can be as simple as a list of preferences. For example, the user can indicate whether he wants the browser to silently reject some types of action without an appropriate permission, or to prompt the user to manually override exceptions on a case-by-case basis. Finally, the web browser translates the user's specification and configuration into a mandatory resource access control policy.

At an individual principal level, a principal may specify a discretionary policy regarding how other principals may access its resources, i.e., discretionary resource access control. For example, a principal may allow its child principals to have read permission of its display area, but may not allow its grandchild principals to have the same access.

D. Resource Policy Enforcement

Following the resource access control policies, the resource reference monitor maintains each principal's access rights over resources. Any resource access must be permitted by the resource reference monitor. The policies are written in a rule-driven language that we consider in the next section.

III. Language

We now outline what the policy language might consist of. A policy handles three types of objects: **principals**, **resources**, and **actions**. It defines **rules** that decide whether a principal can take specific actions on a resource. We call a (principal, action, resource) triple a **permission**, and the policy's job is to consider each permission and either **grant** or **deny** it.

A. Principals

In our policy language, a principal is merely a type of resource. This simplifies reasoning about them: E.g., in considering whether to let one frame navigate another, we have one frame acting as a principal and another as a resource.

B. Resources

A resource has the following properties:

class A standardized name distinguishing the resource by its role, or a sequence of them. Possible classes might include: **document**, **text**, **stylesheet**, **script**, **media** (**image**, **video**, **audio**), **frame**, and **cookie**. For example, an HTML page would have the class **document**, while an image loaded into the page should have both the **media** and **image** classes.

type The MIME type of the resource, when applicable.

protocol, domain, port These collectively define the **origin** of the resource, in the same sense as in the Same-Origin Policy. For Web content, this is the origin from which the content was loaded. For a JavaScript function, this is the origin of the script that defined it. For client objects such as frames, these properties may be undefined.

path Similarly, a resource representing Web content has the path part of the URL as a property.

document A frame (i.e. a principal with the **frame** class) sets this property to the document it's displaying.

parent When a resource is the child of another, this property is set to the parent. For instance, if a document A contains a frame F with a document B loaded, we have that B's parent is F and F's parent is A.

C. Actions

An action could have the following properties:

1) • The principal has the class **script**
   • The principal's **protocol**, **domain**, and **port** match those of the resource
   • *Verdict:* **allow**
2) • The principal has the class **script**
   • *Verdict:* **deny**

Fig. 3. Rules implementing the Same-Origin Policy. In practice, such rules would have stipulations for browser Chrome and other nuances.

1) • The resource has the class **image**
   • The resource's **protocol** is **http** or **https**
   • The resource's **domain** is **images.x.edu**
   • The resource's **port** is 80 or 443
   • *Verdict:* **allow**
2) • The resource has the class **image**
   • *Verdict:* **deny**

Fig. 4. A discretionary policy for a site `www.x.edu` allowing those images, and only those images, served from `images.x.edu`.

class A well-defined name or sequence of them, as for resources. Action classes would be generalizations of HTTP methods, such as *get* or *post*, as well as others such as *call* to call a JavaScript function and *read-cookie* and *write-cookie* for cookie management.

protocol This is also likely specified in a resource, but there may be cases where a resource is loaded using a protocol other than that in its URL (perhaps some sort of caching protocol).

security Information about the security of the action in question. This could allow a policy to deny permissions to load documents using obsolete or compromised encryption algorithms, for instance.

D. Rules

Now that we have defined the three components of a permission—principles, resources, and actions, we use **rules** to reason about it and decide whether to grant or deny it. A rule is a list of **predicates** and a **verdict**. The verdict is either **grant** or **deny**, and a permission matching all the predicates is then granted or denied, respectively. The precise syntax of the predicates is unimportant here; in giving examples, we will simply use English.

E. The Policy

Finally, a **policy** is simply a list of rules. Each rule is tried in order until one of them gives a verdict, and accordingly the permission is either granted or denied.

F. Examples

Many common security mechanisms have natural expressions in this language. E.g., the Same-Origin Policy can be enforced by two simple rules (Figure 3). Discretionary policies, tailored to particular sites, would also be easy to implement. If `www.x.edu` serves all images on its pages from `images.x.edu`, it can use a policy like that in Figure 4.

1) • The action has the class **navigate**
   • The principal is an ancestor of the resource
   • *Verdict:* **allow**
2) • The action has the class **navigate**
   • *Verdict:* **deny**

Fig. 5. A policy specifying that a frame can only navigate its descendants.

IV. Security Effectiveness Analysis

We now demonstrate how our approach can be used to address typical web security problems. We look at three popular web attacks: frame hijacking, cross-site request forgery (CSRF), and DNS rebinding attack. We show as long as the resource management framework is in place, all we need is to specify robust security policies to secure web-based activities.

A. Frame Hijacking

A frame often contains content from sources of different trustworthiness, e.g., hosted advertisements, Flickr albums, Google Maps, or PayPal account pages. Each such content fragment is inside a separate sub-frame. One special access right among frames is the *navigation* right, by which one frame can direct another frame to load its contents from an arbitrary URL. However, it is dangerous if one frame can freely navigate another frame; for example, a malicious frame can send another frame to a phishing site, or replace a gadget in a mashup site with a similar-looking but malicious gadget. Researchers have studied both cross-window and same-window frame navigation attacks [5].

In resource management framework, every frame is a principal, and also a resource (see Section III-A). If a frame wants to navigate another frame, the former as a principal must have the navigation right over the latter as a resource.

The navigation right can be specified either at the browser level or by individual frames. The specified policies then can be enforced by the resource reference monitor. For example, as proposed in [5], the browser may enforce a policy that allows a frame to navigate only its descendants. Fig. 5 shows how this policy might be written in our language.

B. Cross-Site Request Forgery (CSRF)

A CSRF attack occurs when a malicious web page serves a piece of HTML or JavaScript code that causes a web browser to issue a separate request to a target web site, which leads to unintended side effects without the web user's knowledge. A few current practices to prevent this attack include examining the referrer header in the HTTP request, or enforcing shorter expiration time for a session so that a user is likely logged out from a privacy-sensitive web site. But these measures are to be taken on the target web server and require web sites to cooperate. There has not been much that the user of a web browser can do to effectively prevent this attack.

With our resource management framework, we can stop the CSRF attacks by specifying the correct policy. We require that each cookie to be sent in a request pass a permission check by the resource manager. This permission triple will include:

principal The page being accessed.

- The action has class **read-cookie**
- The action's **referrer** either is not set or does not have an origin matching that of the principal
- *Verdict:* **deny**

Fig. 6. A rule preventing CSRF by allowing some cookies to be read only when following internal links.

resource The cookie to be sent to the page.

action An action object with the **class** property being **read-cookie** and the **referrer** property being the page that provided the link.

Given these provisions, a policy could use the rule in Fig. 6 to filter certain cookies, such as security-sensitive login credentials, from being passed along when a cross-site request occurs. In fact, a similar rule could be applied to any resource that we want to avoid sending across origin boundaries.

For instance, an attacker's web page includes a cross-site request `<img src=bank.com/transfer?to=Mallory&amount=100>`. Even if the victim is logged onto `bank.com` and has an active session, that cookie will be filtered out and the transaction will be forbidden. Normal clicks on links in the bank's own pages, however, will function as usual.

C. DNS Rebinding Attack

A DNS rebinding attack happens as follows: First, a malicious web site, say `attacker.com`, answers DNS queries for `attacker.com` with the IP address of its own server, and serves visiting clients malicious JavaScript. Then, when the script sends another DNS query for `attacker.com`, the attacker will return the IP address of a victim server instead. The browser will treat the victim server to have the same origin as `attacker.com`, and the script therefore can read freely from the victim. A common solution to this attack is DNS pinning: a browser should always return the same IP address for a given host name. DNS pinning, however, has been hard to enforce because a browser and its plug-ins all have its own pinning record, leading to a multipin vulnerability.

A suggested solution is to have a database of host name to IP pinnings shared by a browser and all its components [7]. With our resource management framework, implementing this solution is straightforward: We can include this database as a global resource available to all principals. When resolving a host name, a principal will check this database first; if the name is already pinned to an IP address, no DNS queries will be sent out, effectively addressing the multipin vulnerability.

V. Discussion

Our resource management framework is general enough that both a newly designed browser and an existing browser can consider supporting the framework. Individual browsers can implement it differently while considering cost and performance overhead. It may also be possible to implement the framework as a library or micro-kernel for a browser to consult for all its resource-management-related operations.

Incremental deployment is also easy. Call web browsers and sites that support our approach *new* browsers and sites, respectively. If a new browser visits a site, it will simply assume the latter does not implement discretionary resource access control, and continue to apply its mandatory resource access control policies and the discretionary resource control policies from new sites. If an old browser visits a new site, the browser will not recognize the special language for specifying discretionary access controls and will simply ignore it.

Our resource management framework requires precise specification for its resource access control policies. While in this paper we outlined the basic structure and semantics of the language, a more concrete policy specification language will be needed in order for a web browser to understand a web site's resource access control policy.

VI. Conclusions

As web browsers have evolved from static content viewers to a common environment often shared by multiple principals (documents, JavaScript code, plug-ins, and other web components) from different origins, the security of the web browser, and even the Web itself, becomes shaky. While security patches have been distributed and new browser architectures have also been proposed, our position is that a web browser should be treated as a resource allocator in order to secure principals from one another and secure web-based operations.

In this paper we proposed a resource management framework that a web browser can use to control the access of various resources by any principal. Not only a user or a system administrator can enforce a mandatory resource access control policy at the browser level, but the writer of a web page can also specify its discretionary resource access control policy. With the security policies enforced by a resource reference monitor, we further showed that web attacks can be successfully prevented, as exemplified by the prevention of the frame hijacking attacks, cross-site request forgery, as well as the DNS rebinding attack.

Our resource management approach to web browser security is advantageous in that the framework is general enough for various browsers to implement, is incrementally deployable, and is effective and light-weight in addressing new web attacks because of the ease to install new security policies.

References

[1] A. Barth, C. Jackson, C. Reis, and Google Chrome Team, "The security architecture of the Chromium browser," Stanford Univ., Tech. Rep., 2008.

[2] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The multi-principal OS construction of the Gazelle web browser," Microsoft Research, Tech. Rep., 2009.

[3] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A safety-oriented platform for web applications," in *Proc. of the IEEE Symposium on Security and Privacy*, 2006, pp. 350–364.

[4] C. Grier, S. Tang, and S. T. King, "Secure web browsing with the OP web browser," in *Proc. of the IEEE Symposium on Security and Privacy*, 2008, pp. 402–416.

[5] A. Barth, C. Jackson, and J. Mitchell, "Securing frame communication in browsers," in *Proc. of the USENIX Security Symposium*, 2008, pp. 17–30.

[6] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, "SOMA: mutual approval for included content in web pages," in *Proc. of the Conference on Computer and Communications Security*, 2008, pp. 89–98.

[7] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from DNS rebinding attacks," in *Proc. of the Conference on Computer and Communications Security*, 2007, pp. 421–431.