Capability-Based Access Control for Web Browsers

Tongbo Luo and Wenliang Du
Department of Electrical Engineering & Computer Science
Syracuse University, Syracuse, New York, USA

ABSTRACT

Over the last two decades, the Web has significantly transformed our lives. Along with the increased activities on the Web come the attacks. A recent report shows that 83% of web sites have had at least one serious vulnerability. As the Web becomes more and more sophisticated, the number of vulnerable sites is unlikely to go down. A fundamental cause of these vulnerabilities is the inadequacy of the browser's access control model in dealing with the features in today's Web. We need better access control models for browsers.

Today's web pages behave more and more like a system, with dynamic elements interacting with one another within each web page. A well-designed access control model is needed to mediate these interactions to ensure security. The capability-based access control model has many properties that are desirable for the Web. This paper designs a capability-based access control model for web browsers. We demonstrate how such a model can be beneficial to the Web, and how common vulnerabilities can be easily prevented using this model. We have implemented this model in the Google Chrome browser, and have conducted case studies and evaluation on our design and implementation.

1. INTRODUCTION

Over the last two decades since the Web was invented, the progress of its development has been tremendous. From the early day's static web to today's highly dynamic and interactive Web 2.0, the Web has evolved a lot, and there is no sign that the evolution will slow down in the next decade: with the highly expected arrival of HTML5 and the so-called "Web 3.0", the Web will become more and more powerful, sophisticated, and ubiquitous.

During the technology evolution, security has often been overlooked. In the early days of the Web era, attacks on the web were not many. However, a recent report produced by WhiteHat Security indicates that 83% of websites *have had* at least one serious vulnerability, 64% *currently have* at least one serious vulnerability, and the average number of serious vulnerabilities per website is 16.7 [24]. These numbers are quite alarming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2011 Syracuse University.

It is very tempting to blame the developers of web applications for these vulnerabilities, because the vulnerabilities are indeed caused by their mistakes. However, this blame cannot explain an obvious fact: the percentage of vulnerable websites is way too high, much higher than what we have seen in traditional software. Although mistakes are made by developers, something more fundamental might have caused such a significant increase in mistakes.

To find out the fundamental causes, we need to look at the evolution of the Web. Figure 1 depicts the change of the Web during the evolution. A clear trend highlighted in the figure is the evolution of web contents from static to dynamic. With more and more dynamic entities included in web pages, and more and more interaction among these entities, a web page starts to behave like a system. If these entities are not equally trusted, their interactions must be mediated, and thus a well-designed access control system is necessary to mediate the interactions within each web page.

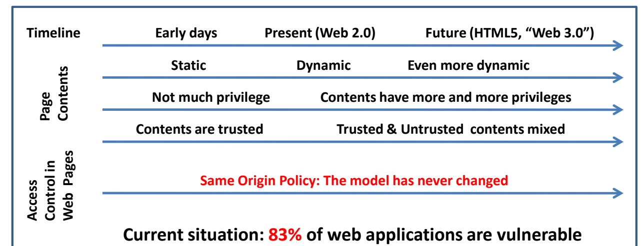

Figure 1: The Evolution of the Web

Unfortunately, there is no such an access control system in the current Web. In terms of access control, the Web has adopted a policy called Same-Origin Policy (SOP). This policy protects entities of one origin from those of others. It does not provide access control on the interaction within a page. Such a protection is unnecessary in the early day's Web, because contents were mostly data. When dynamic contents were first introduced, most contents then were equally trusted because they were generated mostly by the websites themselves. Today's web has a totally different picture. A web page can simultaneously contain entities with varying levels of trustworthiness. For example, advertisement entities from a third-party and entities provided by users are less trustworthy than the entities provided by the websites.

Without an in-page access control system in browsers,

there is no way to directly mediate the interaction among these entities. Web application developers are forced to find alternatives. A common alternative is to conduct validation at the server side before putting untrusted entities in a web page. For example, validation can attempt to remove dynamic entities, disable dynamic entities, or restrict their behaviors. The objective of the validation is to conduct the control at the server side, so when the entities arrive at the browser side later, no undesirable access is possible. This approach is quite awkward, because in typical access control, control is conducted when the access has already been initiated. However, this alternative conducts "control" before any access is initiated. Because the actual accesses are unknown, developers have to infer what potential accesses are from the contents. This inferring process is quite error-prone, and has contributed to a large portion of the high percentage of vulnerabilities in web applications [24]. The best solution is to conduct access control after the access action is already initiated and thus becomes known, but the current web does not have such an access control system. That is a design mistake, and this mistake is one of the fundamental causes of the security problems in the Web.

As the Web is still evolving, it is not too late to fix this design problem. Actually, there are already efforts toward this goal. There are two types of approaches: one approach is to propose specific features to incrementally build an access control system for the web. This approach have resulted in various proposals [6, 9, 12, 13, 15, 17, 18, 22], and the features that are proven to be good will eventually be adopted. Another approach takes a holistic view: it treats the task as developing a complete access control system for web browsers, not as developing pieces for such a system. Once this becomes the goal, there are lot of things that we can learn from, such as the access-control design in operating systems, databases, and many other computer systems.

A representative work of the holistic approach is the Escudo work [11]. Based on the special needs in web applications, Escudo proposes a ring access control for web browsers. Escudo is a start towards designing a good access control system for browsers, but there are needs that cannot be easily satisfied by Escudo. Looking at the evolution history of access control systems in operating systems, one lesson that we have learned is that one model may not be able to fit all the needs. In current operating systems, many models coexist. For example, in `Linux`, Access Control List (ACL), Capability, and Role-Based Access Control (RBAC) coexist; in `Windows 7`, ACL, Capability, and Multi-level Access Control coexist. These models jointly address the different protection needs in operating systems. The fact that these particular models are chosen is the results of many years of evolution in operating systems. We strongly believe that the Web will and should go down a similar evolution path; sticking to the current SOP model prevents us from starting this evolution path.

Motivated by the evolution of access control in operating systems, and by the shortcomings of SOP and Escudo, we decided to study another model that has been widely adopted by modern operating systems. This is the capability-based access control. The main idea of capability is to divide the privileges in a system into smaller pieces, so they can be assigned to the tasks based on the privileges they need. The capability allows us to follow the principle of least privileges, one of the essential principles in designing security systems.

The benefit of having a capability-based access control model in browsers is three-fold. First, by dividing the privileges of web contents into smaller pieces, web browsers can conduct a a finer-grained access control. Second, because the privileges are divided into smaller pieces, web application developers can assign different sets of small privileges to the contents with different levels of trustworthiness. With this model, web developers do not need to conduct the complicated and error-prone process to filter out dangerous contents from the untrusted contents; instead, they can simply assign less privilege (or no privilege at all) to the contents that are not so trustworthy. This is the essence of the least-privilege principle. Third, the capability model allows the system to dynamically adjust the privileges based on environment conditions. For example, if a web page is embedded into a third-party `iframe`, or is overlapping with another `iframe` (a technique used in clickjacing attacks [7]), the page can require the browsers to downgrade the privileges for safety purpose. These three benefits will be discussed in more details in the rest of the paper.

2. ACCESS CONTROL MODELS

2.1 The Needs

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

Principals in a web page are elements that can initiate actions. There are several types of principals inside a web page. (1) Dynamic contents, such as Javascript code, are obvious principals. It should be noticed that Javascript code can be invoked in many different ways: through an embedded *script* tag, being triggered by events, such as *onload*, *onmouseover*, and time. (2) Many HTML tags in a web page can initiate actions, such as `a`, `img`, `form`, `iframes`, `button`, `meta` [1], etc. These tags are considered as principals. (3) Plugins can also initiate actions, so are also considered as principals. However, since plugins usually have their own built-in access control mechanisms, this type of principals is beyond the scope of this paper.

Objects include everything in a web page or those associated with a web page. Web browsers represent the internal contents of a web page using a hierarchical data structure called Document Object Model (DOM), and principals can use DOM APIs to access the objects (called DOM objects) in a web page. DOM objects are obviously considered as objects in our access control system. Cookies are another type of objects. Although they are not included in a web page, they are associated with the web pages from the same domain. Principals can access/modify cookies.

[1] The `meta` tag is supposed to be put in the header of a web page only, but most web browsers accept it if it is embedded in the body of the page. The `Set-Cookie` attribute in the `meta` tag can change cookies.

Modern web applications are quite complicated. Typically, a server-side script combines data and programs from several sources to create a web page. As a result, a web page is composed of several principals and objects with varying levels of trustworthiness, and a proper access-control model must recognize and support this diversity. Some portions of the web page may contain user-supplied contents; principals arising from such HTML excerpts should have limited privileges. For example, consider a blog application: a web page may display a blog post with comments from other users. The original blog post and the comments from users should be isolated from one another so that a deftly constructed malicious comment cannot affect the original blog post.

2.2 The Escudo's Ring Model

Escudo introduces a `ring` concept, which is borrowed from the Hierarchical Protection Rings (HPR) [20] access control model. Rings in Escudo are labeled 0, 1, $\ldots$, N, where N is application dependent and defined by the developers of web applications. In the HPR model, higher numbered rings have lesser privileges than lower numbered rings; namely, ring 0 is the highest-privileged ring, and ring N is the least-privileged ring.

Escudo places all the principals and objects in a web page into these rings based on their trustworthiness. To achieve this, Escudo introduces an attribute called `ring` for the `<div>` tag to assign a ring label to each `div` region. Escudo-enabled browsers will then enforce a simple access control rule based on these ring labels: principals at the ring p can only access the objects at rings o if $p \leq o$. This rule prevents the less trustworthy principals from accessing (read and modify) the more trustworthy contents.

2.3 Capability Model

There are two types of privileges within a web page. One is the privileges to access certain objects (DOM objects and cookies), we call these privileges the *object-based privileges*. The other type is the privileges to access certain actions, such as invoking AJAX APIs, issuing HTTP POST requests, accessing cookies, etc. Whether a principal can access these actions or not has security consequences. We call this type of privileges the *action-based privileges*. Escudo can deal with the object-based privileges quite well, but it is inappropriate for controlling the action-based privileges, because no specific objects are associated to the action-based privileges. As a result, a principal in Escudo basically has all the action-based privileges entitled to its origin, regardless of which ring it is in. This is a clear violation of the least-privilege principle: if a Javascript code from a semi-trusted third party only needs to send HTTP GET requests within the page, we should not give this code the privilege to invoke AJAX APIs or send HTTP POST requests.

With the evolution of the Web, many new action-based privileges will be introduced. AJAX is such an example, it is a newly introduced feature for the Web; being able to conduct AJAX is therefore a new action-based privilege. HTML5 introduces many more action-inducing tags, such as the `<canvas>` and `<video>` tags. These tags increase the attack surface of HTML5-enabled web applications. Therefore, the privileges to initiate these new HTML5 actions should not be given to every principal.

To secure the Web, controlling the uses of action-based privileges must be built into the browser's access control model. This is not the first time that we face this issue; operating systems encountered the same issue long time ago. In operating systems, many applications require action-based privileges. For example, the `ping` program in `Unix` requires the privilege to use raw sockets, the system backup programs require the privilege to read all the files, etc. These programs use to be `setuid` programs, i.e., when they are running, they have all the privileges of the `root` account. This is clearly a violation of the *least-privilege* principle. The modern operating systems solved this problem using the capability concept. The main idea of capability is to define a "token" (called capability) for each privilege; a principal needs to possess the corresponding tokens if it wants to use certain privileges. Because of these fine-grained capabilities, we can assign the least amount of privileges to principals.

The Web has evolved to a stage where it becomes too risky to assign all the action-based privileges to the principals within a web page. These privileges should be separated, and assigned to principals based on their needs and trustworthiness. The same-origin policy model does not separate these privileges, neither does Escudo. To handle the web applications with ever-increasing complexity and to reduce the risks of web applications, we believe that web browsers should adopt the capability model in its access control. As a first step towards this direction, we have designed a capability-based access control for web browsers; we have implemented our design in Google Chrome, and have conducted case studies using our implementation.

3. CAPABILITY FOR BROWSERS

There are three major components in a capability-based access control: the list of capabilities supported by the system, how capabilities are binded to principals, and how access control is enforced. We will discuss each of these components in this section.

3.1 Capabilities

Learning from the history of capability design in `Linux`, we know that the development of capabilities is an evolving process: in this process, rarely used capabilities may be eliminated, more desirable capabilities may be added, new privileges may be introduced when the system evolves, and so on. Therefore, we do not intend to come up with a list of capabilities that are complete. We consider our efforts of introducing capabilities in web browsers only as the first step in such an evolution. In this initial step, we have identified a list of capabilities [2]. They are classified into five categories:

- *Capabilities to access sensitive resources*, including bookmarks, Cookies, Certificates, HTML5 LocalStorage, and Custom protocol handlers.
- *Capabilities to access history resources*, including Web Cache, History, Downloaded items, Search box terms.
- *Capabilities to access DOM elements*, such as whether a principal is allowed to access DOM objects, register an event handler, or to access the attribute settings of DOM objects.
- *Capabilities to send HTTP Requests*, including Ajax GET/POST and HTTP GET/POST requests.

[2]Not all features in HTML5 are included in this paper, as HTML5 is still a work in progress.

- *Capabilities to run Javascript programs or plug-in programs*, including Flash, PDF, Video, Audio, etc.

As a proof of concept, we have only implemented a subset of the above capabilities in our prototype, including capabilities to set cookies, read cookies, use cookies (i.e. attaching cookies to HTTP requests), capabilities to send AJAX GET/POST requests, capabilities to send HTTP GET/POST requests, and capabilities to click hyperlinks and buttons. In our system, we use a bitmap string to represent capability lists, with each position of the bitmap string representing one specific capability. Figure 2 illustrates the specification of the bitmap.

Figure 2: Capability Bitmap

3.2 Static Binding of Capabilities

To use capabilities in access control within a web page, web developers, when constructing a web page, need to assign capabilities to the principals in the page, based on the actual needs of principals. As we have discussed before, principals are DOM elements of a web page. In Escudo, the HTML `div` tag is used for assigning the ring label to each DOM element. HTML `div` tags were originally introduced to specify style information for a group of HTML tags; Escudo introduces a new attribute called the `ring` attribute for the `div` tag. To be consistent with Escudo, we take the same approach. We add another attribute called `cap` for the `div` tag. This attribute assigns a capability list to all the DOM elements within the region enclosed by its corresponding `div` and `/div` tags. An example is given in the following:

```
<div cap="110001111">
   ... contents ...
</div>
```

In the above example, the privileges of the contents within the specified `div` region are bounded by capabilities 1, 2, 4, and 6; namely no DOM elements within this region can have any capability beyond these four.

3.3 Dynamic Binding of Capabilities

Although the actual capability bindings are carried out at the browser side, the static bindings are already decided at the server side by the server-side programs. Such a decision does not consider any environment condition at the browser side, as the decision is already finalized before web pages is sent to browsers. Therefore, static bindings do not allow web applications to adjust the privileges of principals based on the environment conditions of the browser. We use a case study to demonstrate why knowing the environment conditions is important to security.

ClickJacking. ClickJacking [7] is one of the attacks that many web applications are facing. The attack does not exploit any vulnerability of web applications, but instead, it exploits the transparent `iframe` feature in web browsers. In ClickJacking attacks, attackers load the target web page (say `gmail.com`) into an `iframe` in their own pages (say `evil.com`). Attackers make this iframe transparent (invisible) to users, and overlay it with an visible iframe page. Attackers then lure victims to visit their web site. These two iframes can be set up in a way such that when victims click in the area within the visible iframe, the actual click goes to the invisible iframe. The above setup creates an visual illusion to users, because what the victims have actually done is different from what they think they have done. As a consequence in the above example, users can be tricked into clicking a sequence of buttons in the visible iframe that lead to the deletion of the emails in their Gmail accounts (in the invisible iframe) [3].

Although the ClickJacking problem can be defended by various solutions [8,14,25], those solutions are quite specific to the ClickJacking problem alone. They are not general solutions: if some other features like the transparent iframe is introduced in browsers, we have to come up with other specific solutions. We need to think about what fundamental changes that we can make, so we can systematically deal with a class of problems that come up due to the introduction of new features in web browsers. From the access control perspective, the ClickJacking problem is caused by the failure of knowing the browser-side environment condition by the server-side programs. This is a dilemma of the static binding approach, because the privileges of the contents of a web page has to be decided before the page is sent to browsers.

Dynamic bindings can solve the ClickJacking (and alike) problem in a fundamental way. Dynamic bindings allow the binding of privileges to principals to be based on the environment condition on the browser side. For example, to solve the ClickJacking problem, we can specify the following dynamic binding policies:

```
if (not in a iframe)
    Bind capabilities 0, 1, 3, 4,
else if (embedded in a non-overlapping iframe)
    Bind capabilities 0, 3
else if (embedded in overlapping iframe)
    Bind no capability
```

The basic idea of the above example is to reduce the capability assigned to principals depending on whether the web page is embedded in an iframe. If the iframe does not overlap with other iframes (or other contents of the hosting web page), the privilege of principal is downgraded a little bit; if the iframe does overlap with other contents, all privileges of the principal are taken away.

The above dynamic binding policy can be implemented using bitmap masks. We introduce two bitmap masks. The first is called `iframemask`, specifying which capabilities can be assigned to principals if the page is loaded into an iframe of a third-party web page. The second mask is called `overlapmask`, specifying which capabilities can be assigned to

[3]The attack will only succeed if the victim have an active session with Gmail.

principals if the page is loaded into an iframe that overlaps with other contents in the page. The actual capability list is `cap & iframemask` or `cap & iframemask & overlapmask`, respectively, where `&` is a bit-wise `AND` operation. The above policy example can be specified in the followings:

```
<div cap="11011" iframemask ="10010"
                 overlapmask="00000">
   ... contents ...
</div>
```

3.4 Capability Enforcement

Enforcement in capability-based access control is well defined in the capability model: an access action is allowed only if the initiating principal has the corresponding capability. The main challenge in a capability system is to identify the initiating principals and their associated capabilities. In general, identifying principals is not so difficult: whenever an action is initiated (either by Javascript code or by HTML tags), the browser can easily identify the `div` region of the code or tags, and can thus retrieve the capabilities binded to this region. Unfortunately, as a proverb says, the devil is in the details; identifying principals is quite non-trivial. We describe details of capability enforcement in the Implementation section.

4. ENSURING SECURITY

The key to capability enforcement is the integrity of the configuration (i.e., capability assignment) provided by the application. We describe additional measures to prevent the configuration from being tampered with.

Configuration Rule: Protecting against Node Splitting. Any security configuration that relies on HTML tags are vulnerable to node-splitting attacks [12]. In a node-splitting attack, an attacker may prematurely terminate a `div` region using `</div>`, and then start a new `div` region with a different set of capability assignments (potentially with higher privileges). This attack escapes the privilege restriction set on a `div` region by web developers. Node-splitting attacks can be prevented by using markup randomization techniques, such as incorporating random nonces in the `div` tags [6, 17]. Capability-enhanced browsers will ignore any `</div>` tag whose random nonce does not match the number in its matching `div` tag. The random nonces are dynamically generated when constructing a web page, so adversaries cannot predict those numbers before they insert their malicious contents into a web page.

Scoping Rule. When contents are from users, they are usually put into `div` regions with limited privileges. However, user contents may include `div` tags with the capability attributes. If web applications cannot filter out these tags, attackers will be able to create a child `div` region with arbitrary capabilities. To prevent such an privilege escalation attack, we define the following `scoping rule`:

> **Scoping Rule:** The actual capabilities of a DOM element is always bounded by the capabilities of its parent.

Formally speaking, if a `div` region has a capability list L, the privileges of the principals within the scope of this `div` tag, including all sub scopes, are bounded by L. See the following example (note that nonces are used for protecting against node-splitting attacks):

```
<div id="A" cap="10101" nonce=893232>
      ... contents ..
      <div id="B" cap="11111" nonce=932398>
           ... contents ...
      </div  nonce=932398>
</div nonce=893232>
```

In the above example, the `div` region A is the parent of another region B, so B's actual capabilities is bounded by A's capabilities ("10101"), even though B's capability attributes says "11111". Therefore, if A does not have a capability, B will not have it either, regardless of whether the capability attribute of B includes that capability or not.

Access Rule for Node Creation/Modification. Using DOM APIs, Javascript programs can create new DOM elements or modify existing DOM elements in any `div` region. To prevent a principal from escalating its privileges by creating new DOM elements or modify existing DOM elements in a higher privileged region, we enforce the following access rule:

> **Access Rule:** A principal with capabilities L can create a new DOM element or modify an existing DOM element in another `div` region with capabilities L' if L' is a subset of L, i.e., the target region has less privilege than the principal.

Cross-Region Execution Rule. In most web applications, Javascript functions are often put in several different places in the same HTML page. When a Javascript program from one `div` region invokes a function in another `div` region, what should be considered as the principal, and whose capabilities should be used? A simple design is to only treat the initiating program as the principal, and use its capabilities in access control. A downside of this design is that when the invoked function is in an area less trustworthy (i.e. having less privileges) than the invoking program, the function will be actually invoked with higher privileges than what it is entitled to.

To avoid the above situation, a modified design is to allow a Javascript program with privilege $\mathcal{A}$ to invoke a function with privilege $\mathcal{B}$, only if $\mathcal{A}$ is a subset of $\mathcal{B}$. Namely, no Javascript can invoke a function with less privilege. This way, we can always use the initiating program's privilege $\mathcal{A}$ throughout the entire execution.

A more general solution is to allow the invocation regardless of what relationship $\mathcal{A}$ and $\mathcal{B}$ has, but ensure that when the function is invoked, the privilege of the running program becomes the conjunction of $\mathcal{A}$ and $\mathcal{B}$, i.e., $\mathcal{A} \wedge \mathcal{B}$. Namely, the privilege will be downgraded if $\mathcal{B}$ has less privilege than $\mathcal{A}$. After the function returns back to the caller, the privilege of the running program will be restored to $\mathcal{A}$ again. We have implemented this general solution in our prototype.

5. IMPLEMENTATION

5.1 System Overview

In Google Chrome [4], there are four major components

[4] We use Version 3.0.182.1, simply because this is the version we had when we started the implementation. We plan to port our implementation to the most recent version.

closely related to our implementation: Browser Process (Kernel), Render Engine, Javascript Interpreter (V8), and sensitive resources. We add two subsystems to Chrome: *Binding System*, and *Capability Enforcement System*. Figure 3 depicts the positions of our addition within Chrome.

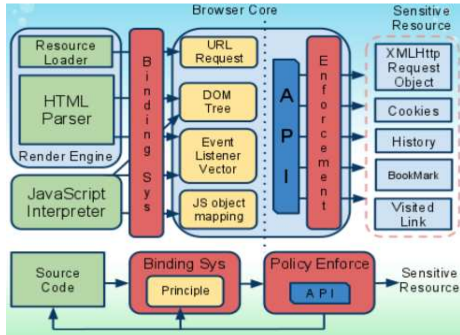

Figure 3: System Overview

Binding System. The purpose of binding system is to find the capabilities of a principal, and store them in data structures where the enforcement module can access. In Chrome, principals are identified in several components, including HTML parser (which parses the HTML code and generate DOM tree) and Javascript Interpreter (which compiles Javascript code into V8 objects). We need to modify those components to bind capabilities to principals when principals are created.

The capability information of each principal is stored inside the browser core. Only the browser's code can access the capability information; the information is not exposed to any code external to the browser's code base (e.g. Javascript code). At this point, we do not foresee any need for the external code to access such information, so no API is provided by the browser for the access of the capability information.

Effective Capabilities. When an access action is initiated within a web page, to conduct access control, the browser needs to know the corresponding capabilities of this access. We call these capabilities the *effective capabilities*. Identifying effective capabilities for actions is nontrivial: in the simplest case, the effective capabilities are those attached to the principal that initiates the action. Javascript code makes the situation much more complicated. A key functionality of our binding system is to keep track of the effective capabilities within a web page. We will present the details later in this section.

Capability Enforcement. Our implementation takes advantage of the fact that every user-level access to sensitive resources goes through the browser kernel, precisely, through the local APIs. For example, `AssembleRequestCookies()` in the `URLRequestHttpJob` class will be invoked to attach cookies to HTTP requests; `XMLHttpRequest::send()` will be called for sending Ajax requests. We add the capability enforcement to these APIs. When they are called, the enforcement system checks whether the effective capabilities have the required capabilities for the invoked APIs.

Actions. Within a web page, there are two types of actions: (1) *HTML-induced actions:* this type of actions are initiated by HTML tags, such as the HTTP requests caused by `<img>`, `<iframe>`, `<meta Set-Cookie>`, `submit` buttons, etc. (2) *Javascript-induced actions:* this type of actions are initiated by Javascript code. For both types of actions, when they take place, we need to identify the *effective capabilities* that should be apply to the actions.

5.2 HTML-Induced Actions

The effective capabilities of HTML-induced actions are the capabilities assigned to the `div` region that the initiating HTML tags belong to. When a web page reaches a browser, it will be parsed by the browser's HTML parser. A main function of the parser is to generate a DOM tree, and the contents of a web page will be placed in DOM objects in the tree. The `div` regions will be represented as DOM objects.

The capability attributes introduced by us will be treated by the parsers as attributes of a tag, just like any other attribute. After extracting the capability attributes, the HTML parser will pass the information to our binding system, which maintains a *shadow DOM* tree. This shadow DOM tree stored the capabilities of each DOM node, and it can only be access by our binding system. Although Javascript programs can modify the attributes of DOM objects through various APIs, these APIs cannot be used to modify the capability attributes, as the values of the capability attributes are stored in the shadow tree, not the original tree. No API is exposed to Javascript programs for accessing the shadow tree.

When an action is initiated from a HTML tag, the enforcement system identifies the DOM object that the tag belongs to, retrieves the capabilities from its shadow object, and finally checks whether the capabilities are sufficient to carry out the action or not. If not, the action will not be carried out.

5.3 Javascript-Induced Actions

Identifying the effective capabilities of Javascript-induced actions is quite complicated. This is because a running sequence of Javascript can span *multiple* principals with different sets of capabilities. For example, the execution may start from the Javascript code in one `div` region, but the code can invoke Javascript functions in other `div` regions. In this case, the cross-region execution rule described in Section 4 will apply. For example, if A calls B, B calls C, and C calls D, then when executing D, the effective capabilities are the conjunction of the capabilities of A, B, C, and D. When function D returns to C, the effective capabilities will become the conjunction of the capabilities of A, B, and C.

We use a stack data structure in our binding system to store the effective capabilities in the runtime (we call it the capability stack). When a Javascript program gets executed, the capabilities of the corresponding principal will be pushed into the stack as the first element of the stack. The top element of the stack is treated as the *effective capabilities*, denoted as E. When a function in another principal (say principal B) is invoked, the updated effective capabilities $E \wedge Cap(B)$ will be pushed into the stack, where $Cap(B)$ represents the capabilities assigned to B. When the function in B returns, the system will pop up and discard the top element of the stack.

The capability stack must be updated every time the principal of code changes during the execution. Therefore, our binding system must be involved when the principal of execution changes. Since the invocation of functions happens

insider the Javascript Interpreter (the V8 engine), the ideal solution is to build part of the binding system in V8: when the HTML parser sees Javascript code, it identifies the capabilities of the principal, and pass them into V8. This way, each function object within V8 is attached with a capability list; when a function is invoked, V8 can push the effective capabilities into the capability stack.

The situation is further complicated by another fact: V8 compiles Javascript code into native code at run-time; therefore when a function invocation happens, it may not go through the V8 engine, and thus our binding system cannot be triggered to update the capability stack.

An alternative solution is to not modify the V8 engine, but instead to modify the Javascript code. We introduce a code rewriting module, which rewrites code before sending it to the V8 engine. The rewritten code first pushes the effective capabilities of the next running principal into the capability stack before executing the invoked function, and pop the top element from the capability stack right after the current function returns. To accomplished this goal, we introduce two built-in Javascript functions:

```
void Cap_Push(capability, random_number);
void Cap_Pop(random_number);
```

Because the invocation of these two built-in Javascript function can change the runtime effective capabilities, we cannot allow user's code to call these functions; we can only allow our rewriting module to add the invocation of these two functions into Javascript programs. To achieve this goal, we pass a random number to these two functions. This number is generated by our rewriting module for each page. When `Cap_Push` and `Cap_Pop` are invoked, the numbers in their arguments must match with the random number held by the browser kernel for that page. Since the contents of rewritten code are invisible to Javascript code, and the random number is only known to the browser, it will be hard for attackers to guess this random number; any invocation with a mismatched number will cause the invocation to return without doing anything.

The following code gives an example on how the rewriting module wraps the function `foo()`:

```
Original function:
   function foo(arg) { /*function body*/ }

Rewritten function:
   var _tempAOP_12453 = foo;
   foo = function(arg){
      try{      Cap_Push('10101',83940);
                return _tempAOP_12453.apply(this, arg); }
      finally{ Cap_Pop(83940); }
   }
```

The call `_tempAOP_12453.apply()` will basically invoke the original function `foo`. The code in the `finally` clause will be invoked upon the finish of the invocation.

5.4 Event-Driven Actions

Some Javascript-induced actions are triggered by events, not directly by principals. When these actions are triggered, we need to find the capability of the responsible principals. There are three types of events: DOM-registered events, timer events, and AJAX callback events.

DOM-Registered Events. In browsers, it is possible to register handlers for specific event types and specific DOM nodes. Whenever the specified event occurs to the registered DOM nodes, the handler for that event, if any, is called. In this situation, we need to identify the responsible principals and their associated capabilities.

There are two ways to register handlers. One way is to do it statically through HTML tags/attributes, and the other is to do it using Javascript. In the static method, event handler is specified using HTML event attributes, such as `onclick`, `onload`, `onclick`, etc. The following HTML excerpt registers a block of code as an event handler to a button; when the button is clicked, the code will be triggered:

```
<button onclick=" ... code ... ">
```

Another way to register handlers is to use Javascript. To register an event handler (say `onclick`) for a DOM object (say `dom_obj`), we can use the following Javascript code (`clickHandler` is a Javascript function):

```
dom_obj.onclick = clickHandler;
```

Timer Events. Javascript can set timer events using various functions, such as `setTimeout()` and `setInterval()`. These events are not tied to any DOM objects, instead, they are directly tied to the global `window` object:

```
window.setTimeout (code, timeout);
window.setInterval(code, delay);
```

AJAX Callback Events. When AJAX sends out a request, it registers a callback function to the system; the function will be invoked when the response comes back. A typical way to register callbacks is shown in the following:

```
xmlhttp.onreadystatechange
      = function() { /*handler code*/}
```

Binding Capabilities to Event Handlers. Event handlers are triggered by the system (i.e. the browser), not a particular principal. To identify which capabilities should be used when executing the handlers, we need to find out who is responsible for registering the event handlers. The capabilities should be the effective capabilities when the handlers were registered. If they are registered via HTML, such as the `onclick` and `onsubmit` attributes, then the capabilities for the handlers should be those entitled to their containing DOM objects. If they are registered via Javascript, the capabilities for the handlers should be the effective capabilities at the point of registration.

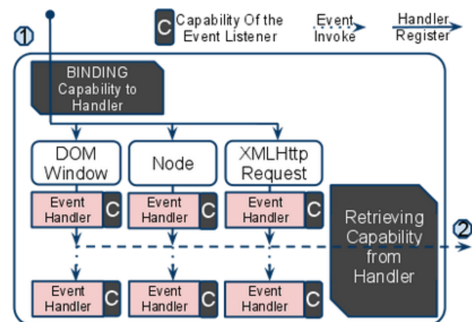

Figure 4: Event Mechanism in Chrome

Figure 4 shows how event registration and triggering work in Google Chrome. Principals that are allowed to register events maintain an "`eventListener` vector" for each type of events. Each item in the vector is an event handler. Each event-register operation should go through the API `addEventListener()` (marked by number 1 in the figure); this API inserts the event handler into the principal's eventListener vector. The invocation of event handler goes through the function `HandleEvent()` (marked by number 2 in the figure).

We modified the `addEventListener()` function, so we can store the effective capabilities into event handlers during the event registration. We also modified the function `HandleEvent()`, so when an event is triggered, we can retrieve the effective capabilities from the event handler objects, and push them into the capability stack.

5.5 Backward Compatibility

Our implementation is backward compatible. There are two scenarios. First, when our modified browser sees a web page without capability attributes, it knows that the page is not enhanced with our capability model, and thus provides all capabilities to the contents, basically going back to the same-origin policy. Second, if a web page is enhanced with our capability tags, but is rendered by a browser that does not implement our capability model, according to the standard, the browser will simply ignore those capability attributes.

6. CASE STUDIES AND EVALUATION

To evaluate how useful, effective, and easy-to-use the capability model is in securing web applications, we have conducted case studies using a number of open-source web application programs, including `Collabtive`, `PhpBB2`, `PhpCalendar`, and `MediaWiki`. For each application, we focus on evaluating the following aspects: (1) defense against Cross-Site Scripting attacks, (2) defense against malicious advertisement, (3) defense against ClickJacking attacks, and (4) limiting the privileges of untrusted inputs. Due to the page limitation, we cannot describe all our case studies in this paper. We will only present several representative ones. Full details will be included in the extended version of this paper.

6.1 The Orkut worm

On 25th September 2010, a new worm affecting Orkut emerged. The basic idea of this worm is to inject a short Javascript code into the victim page using the `onload` event of `iframe`. This code is a "bootstraping" code; its sole purpose is to download and run the attacking Javascript code from another site. Here is the key snippet of the code:

```
<iframe onload="a = document.createElement('script');
        a.src = 'www.malicious.com/malware.js';
        document.body.appendChild(a)"></iframe>
```

Defending against this attack using capability is quite straightforward. Since this block of HTML contents are inputs from users, they should be put in an area that does not have much privilege. For example, they can be put in the following area:

```
<div cap="000000000" nonce="3433893">
   <iframe onload="a = document.createElement('script');
        a.src = 'www.malicious.com/malware.js';
        document.body.appendChild(a)"></iframe>
</div nonce="3433893">
```

Because this region is not given any capability, there are several reasons why the attacks will not be effective. First, because of the lack of HTTP-Request capabilities, no HTTP request can be sent out from this region; therefore, the attacking code cannot be downloaded. Second, even if the region is given the HTTP-Request capabilities, the downloaded code will not gain more privileges; therefore, as long as the cookie-access capability is not given to this region, no effective attack can be launched.

6.2 Untrusted input - AD Network

As a performance-based advertising network, `admedia.com` connects advertisers to consumers across many channels. One of the channels is called affiliating in-text advertising; this is done by importing the following 3rd-party Javascript file into the host page (i.e. the publisher).

```
<script src='http://inline.admedia.com/?count=5&id=OzooNic'>
</script>
```

When visitors browse the host page, they will see the contents of the page as usual; but when they scroll over the linked text, they will be able to see advertisements. The imported 3rd-party Javascript code gets to determine which text will be linked and how often. The Javascript code need to modify the page to achieve this effect.

Since this 3rd-party Javascript code was imported into the host page (publisher), it has the same privilege as those coming from the publisher, i.e., it can do a great damage if the code is malicious. `Admedia.com` claims that the code only adds hyperlinks to the page, so it is against the principle of least privileges if the code is given the privileges beyond what is needed to modify the page; there is no need to allow the code to access cookies, history, etc. Web developers of the host page can limit the privileges assigned to the Javascript code from `Admedia.com` using capabilities. The following example gives the code limited capabilities:

```
<div cap="000001111" nonce="5528053">
   <script src='http://inline.admedia.com/?count=5&id=OzooNic'>
   </script>
</div nonce="5528053">
```

Javascript code from `Admedia.com` is only given four capabilities (HTTP GET/POST and click capabilities), which are sufficient for the code to achieve its purpose. According to the "access rule" discussed before, the code from `Admedia.com` is restricted to access and modify the areas that have equal or less privilege. These should include most of the text areas. If the code is unfortunately malicious, it can deface the web page for sure, but due to the lack of privileges to access cookies, its damage is greatly limited.

6.3 Prevent ClickJacking in PhpBB

To attack PhpBB using the classical ClickJacking attack method, attackers embed the PhpBB web forum into their own web page, putting PhpBB into a transparent iframe that is overlapped with another dummy iframe. When victims click buttons/links in the dummy page, they actually click the ones in PhpBB (e.g. member delete button, message post button). To defeat this attack is quite straightforward using capabilities: the web developers of PhpBB forum can use the dynamic binding to downgrade the privilege of the entire page if the page is embedded in an overlapping iframe:

```
<html>
  <div cap="111111111" overlapmask="111111100" nonce="3996820">
```

```
    ... the entire page ...
  </div nonce="3996820">
<html>
```

Because this `div` region is the outermost region of the page, it is usually assigned all the privileges; its sub-regions will be assigned the privileges based on their needs. However, the `overlapmask` is set to all ones except the last two bits, indicating that the page is not given the hyperlink-click or the button-click capabilities, if the page is loaded by a third-party web site into an overlapping iframe. Basically, the page in the iframe becomes unclickable, so ClickJacking attacks become impossible. Of course, the `overlapmask` can be set to all zeros to drop all the privileges if that is more desirable.

6.4 Prevent XSS in Collabtive

Collabtive is an open-source web-based project management software intended for small to medium-sized businesses and freelancers. This web application provides several channels for users to interact with one another, including message posting, online chatting, project assignment, and user feedback. To prevent Cross-Site Scripting (XSS) attacks, the application has installed many filters and encoding schemes, but still attacks are possible. We can instead use capabilities to defend against XSS attacks.

Modifying Collabtive to benefit from our capability model is quite easy because of the Smarty template [2] used by Collabtive. Because the outputs of web applications are web pages, they have to deal with how to construct web pages using HTML. This is called the view part of web applications. In the past, the view part was often mixed together with the rest of the program logics. Nowadays, thanks to the technologies such as Smarty, web applications can separate the view part from the program logics. For instance, using Smarty, web developers can define a view template file, which contains the majority of HTML code, along with several holes to be filled later by programs.

The assignment of capabilities is done on views. Therefore, if views are already separated from the program logics, assigning capabilities becomes quite simple: we just need to modify the template file. It only took several hours for us to finish the task for Collabtive. The following shows a change we made to a template file called `message.tpl` in Collabtive:

```
<div class="message-in">      <div class="message-in"
                              cap="000000000" nonce={$rand}>
    {$message.text}     ->        {$message.text}
</div>                       </div nonce={$rand}>
```

The `$message.text` area is a hole in the template, and this hole will be filled when the template is used. In Collabtive, `$message.text` will be filled with data provided by users, and no privilege is needed in this hole. Therefore, we assign no capability to this hole. Even if user's inputs contain malicious contents (such as code or action-inducing HTML tags), no damage can be achieved.

6.5 Performance Overhead

To evaluate the performance of our implementation, we have conducted experiments to measure the extra cost our model brings to the Chrome. We measure how long it takes a page to be rendered in our modified browser versus in the original browser. We use some of the built-in tools in Chrome to conduct the measurement. In our evaluation, we tested four web applications: Collabtive, phpBB2, phpCalendar and MediaWiki; we measure the total time spent on rendering pages and executing JS code. The configuration of the computer is the following: Inter(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz, 3.24 GB of RAM. The results are plotted in Figure 5.

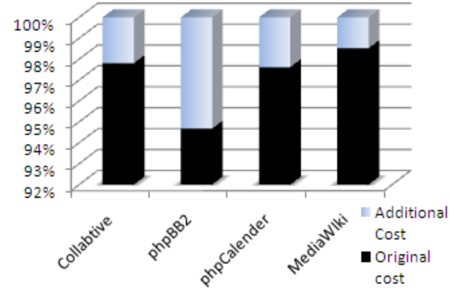

Figure 5: Performance

The results show that the extra cost caused by the model is quite small. In most cases, it is about 3 percent. For phpBB2, it is a little bit higher, because phpBB2 uses more Javascript programs than the others.

7. RELATED WORK

The limitations of SOP have received substantial attention in recent work. Jackson et al. [9] extends the SOP to browser cache content and visited link information to address the problem. Livshits and Ulfar [15] extends the SOP to additionally account for the principal names added to tag groups for neutralizing some XSS and RSS injection attacks. Karlof et al. [13] proposes extending the SOP to account for certificate errors in the origin to distinguish resources in the authentic domain from a spoofed domain to address dynamic-pharming attacks. While each of these proposals addresses a specific shortcoming in the SOP, they do not address the general gap between the SOP's fundamental model and the security requirements of modern web applications. In contrast, our proposed capability model is a fine-grained protection model specifically designed to meet the protection needs of modern web applications.

Another way to overcome the limitation of SOP is to use static and dynamic verifiers to verify the conformance of a Javascript program to a safe subset of the language [1, 4]. The primary target of these tools are applications that embed untrusted and semi-trusted Javascript programs from third parties. Verifiers can be considered as an alternative approach to dealing with the browser's access-control failure. One limitation of this approach is that web applications must trust that the content provider also uses the verifier on their Javascript programs. For example, a web application may lease a portion of its web page to an advertiser; currently it has to trust the advertiser to use a verifier on the Javascript programs provided to display the advertisement. Another limitation of the Javascript verifiers is that these verifiers cannot restrict the access initiated by the HTTP-request issuing principals, which do not involve Javascript programs at all.

Several recent works have proposed new browser architectures. The OP web browser isolates each web page instance

and various browser components using OS processes [5]. Tahoma isolates each instance of a web application inside the browser using separate virtual machines [10]. Essentially, these are two different approaches for isolating web applications from one another and limiting their permissible behavior. Unlike our work, these two approaches are not designed to enforce protection within a web page.

Chromium [3,19] and Gazelle [23] are two new web browsers that use an architecture in which the browser is separated into two portions: kernel and applications. The new architecture has desirable security and reliability properties. However, the access control mechanism is still based on the same-origin policy.

Current work has proposed several solutions for XSS attacks [6,12,17,18,22]. All these solutions are attack-specific patches to the application or browser. In contrast to these solutions that address the symptoms of the underlying problem, our capability model is not a patch for XSS problems. Rather, it is a fine-grained protection model for web browsers suited for modern web applications. XSS problems are thwarted as a side effect of addressing the fundamental weakness in the protection model of web browsers.

Recently, several papers proposed fine-grained security policies for browsers. Content Security Policy [21] intends to provide a security mechanism for web developers to specify how contents interact on their web sites. ConSCRIPT [16] presents a client-side advice system for hosting page to express fine-grained application-specific security policies that are enforced at runtime. Essentially, they introduce capability concepts to web browsers; however, they focus on specifying policies for Javascript code. Our work treats all action-inducing page contents as principals, and enforce capability-based access control on their actions. These principals not only include Javascript code, but also include action-inducing HTML tags. Because of this, our access control model can help solve the ClickJacking problem by putting access restrictions on the `iframe` HTML tag. Capability models for Javascript cannot solve the ClickJacking problem, because the attack does not involve Javascript.

8. CONCLUSION AND FUTURE WORK

To enhance the security infrastructure of the Web, we have designed a capability-based access control for web browsers. This access control model, widely adopted in operating systems, provides a finer granularity than the existing models in browsers. We have implemented this model in Google Chrome. Using case studies, we have demonstrated that many of the hard-to-defend attacks faced by the Web can be easily defended using the capability-based access control model within the browser.

9. REFERENCES

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

[2] Smarty template engine. `http://www.smarty.net/`.

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

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

[5] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P'08)*, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.

[6] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2009.

[7] R. Hansen and J. Grossman. Clickjacking. `http://www.sectheory.com/clickjacking.htm`, 2008.

[8] C. Jackson. Defeating frame busting techniques. `http://crypto.stanford.edu/framebust`, November 16, 2005.

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

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

[11] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS)*, Genoa, Italy, June 21-25 2010.

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

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

[14] E. Lawrence. IEBlog: IE8 Security Part VII: ClickJacking Defenses, February 2009.

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

[16] L. A. Meyerovich and V. B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496, 2010.

[17] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2009.

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

[19] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, New York, NY, USA, 2009. ACM.

[20] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, 1972.

[21] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *WWW*, pages 921–930, 2010.

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

[23] H. Wang, C. Grier, A. Moshchuk, S. King, P. Choudury, and H. Venter. The multi-principal os construction of the gazelle web browser. Microsoft Technical Report MSR-TR-2009-16, February 2009.

[24] WhiteHat Security. Whitehat website security statistic report, 10th edition, 2010.

[25] M. Zalewski. Dealing with UI redress vulnerabilities inherent to the current web. `http://lists.whatwg.org/htdig.cgi/whatwg-whatwg.org/2008-September/016284.html`, September 2008.