Contego: Capability-Based Access Control for Web Browsers

Tongbo Luo and Wenliang Du*

Department of Electrical Engineering & Computer Science,
Syracuse University, Syracuse, New York, USA,
{toluo,wedu}@syr.edu

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 decrease. 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.

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 pages to today's highly dynamic and interactive ones, the Web has evolved a lot. During the technology evolution, security has often been overlooked. 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 [13]. These numbers are quite alarming.

A fundamental cause for many of these problems is the browser's failure to mediate the interactions within each web page. In terms of access control, the Web has adopted a policy called Same-Origin Policy (SOP), which gives all the contents from the same origin the same privileges. Such a policy, sufficient for early day's Web, is not appropriate anymore. The inadequacy of these access control models has been pointed out by various studies [1, 2, 4, 7, 8, 10, 12]. A fundamental problem of SOP is in its coarse granularity. Today's web page can simultaneously contain contents with varying levels of trustworthiness. For example, advertisements from a third party and inputs from

* This work was supported by Award No. 1017771 from the US National Science Foundation.

users are less trustworthy than the first-party contents produced by the websites themselves. These untrusted contents should not be given the same privileges as those trusted first-party contents.

To solve the access control problems faced by SOP, our earlier work, Escudo [5], proposes a ring access control model for web browsers. This model allows web application servers to put contents in different rings, based on their trustworthiness: contents that are more trustworthy are put in the higher privileged rings. Escudo ensures that contents in the lower-privileged rings cannot access contents in the higher-privileged rings. While Escudo has helpped solve a number of security problems on the Web, its granularity on privileges is not fine enough.

There are two typical types of privileges within a web page. One is the privileges to access certain objects, 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 undesirable according to the least-privilege principle: if a Javascript code from a semi-trusted third party only needs to send HTTP GET requests, we should not give this code the privilege to invoke AJAX APIs or send HTTP POST requests.

To secure the Web, controlling the uses of action-based privileges must be built into the browser's access control model. This objective can be achieved using capability-based access control. 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 Contego, a capability-based access control system for web browsers; we have implemented our design in the Google Chrome browser, and have conducted case studies using our implementation. Due to the page limitation, details of the case studies are not included in this paper; they can be found in the extended version of the paper [9].

2 Capability for Browsers

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. (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. Objects include everything in a web page or those associated with a web page. DOM objects are obviously considered as objects in our access control system. Cookies are another type of objects.

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.

2.1 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, 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. They are classified into five categories:

- *Access sensitive resources*: bookmarks, cookies, certificates, HTML5 local storage, and custom protocol handlers.
- *Access history resources*: web cache, history, downloaded item, etc.
- *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.
- *Send HTTP Requests*, Ajax GET/POST and HTTP GET/POST requests.
- *Execute Javascript 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. We use a bitmap string to represent capability lists, with each bit of the bitmap string representing one specific capability.

[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.

2.2 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="11000110"> ... contents ...  </div>
```

In the above example, the privileges of the contents within the specified `div` region are bounded by capabilities 1, 2, 6, and 7; namely no DOM elements within this region can have any capability beyond these four.

2.3 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.

2.4 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 [6]. 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 the markup randomization techniques, such as incorporating random nonces in the `div` tags [3, 11]. Contego-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 a web page is constructed, so adversaries cannot predict them before inserting 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 also include `div` tags with the capability attribute. If web applications cannot filter out these tags, attackers will be able to create a child `div` region with an arbitrary list of capabilities. To prevent such a privilege-escalation attack, Contego defines the following *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.

Access Rule for Node Creation/Modification. Using DOM APIs, Javascript programs can create new or modify existing DOM elements in any `div` region. To prevent a principal from escalating its privileges by creating new or modify existing DOM elements in a higher privileged region, Contego enforces the following access rule: A principal with capabilities L can create a new 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 (with privilege $\mathcal{A}$) invokes a function in another `div` region (with privilege $\mathcal{B}$), Contego enforces the following rule: 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.

3 Implementation

3.1 System Overview

We have implemented Contego in the Google Chrome browser[2]. In Chrome, there are four major components related to our implementation: Browser Process (Kernel), Render Engine, Javascript Interpreter (V8), and sensitive resources. We added two subsystems: *Binding System*, and *Capability Enforcement System*. Figure 1 depicts the positions of our additions within Chrome. We also evaluated our model on multiple web applications; see our full paper [9] for details.

Binding System. The purpose of the binding system is to find the capabilities of a principal, and store them in data structures where the enforcement module can access. 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).

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

[2] We use Version 3.0.182.1. We plan to port our implementation to the most recent version.

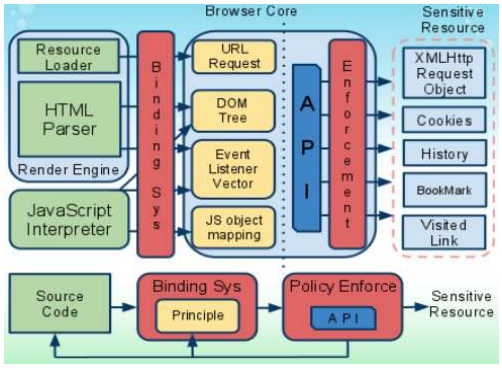

Fig. 1. System Overview

access. We call these capabilities the *effective capabilities*. Identifying effective capabilities 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. When they are invoked, the enforcement system checks whether the effective capabilities have the required capabilities for the invoked APIs.

Actions. Within a web page, there are several types of actions. For each type, when it takes place, we need to identify the *effective capabilities* that should be applied to the actions. We discuss these different types of actions in the following subsections.

3.2 HTML-Induced Actions

Some actions are triggered by HTML tags. The effective capabilities of these HTML-induced actions are the capabilities assigned to the tag's `div` region. These capabilities will be extracted by the HTML parser, delivered to the binding system, and stored in a *shadow DOM* tree. This shadow DOM tree stores capabilities of each DOM node, and it can only be accessed by the 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, because the values of the capability attributes are stored in the shadow tree, not the original tree; no shadow-tree API is exposed to Javascript. When an action is initiated from an HTML tag, the enforcement system identifies its DOM object, retrieves the capabilities from its shadow object, and finally checks whether the capabilities are sufficient. If not, the action will not be carried out.

3.3 Javascript-Induced Actions

Some actions are triggered by Javascript code. Identifying the effective capabilities of these Javascript-induced actions is quite complicated. This is because a running se-

quence of Javascript can span *multiple* principals with different sets of capabilities. The system will trace the executing principal and record the effective capabilities. 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 code 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 code in B returns, the system will pop up and discard the top element of the stack.

3.4 Event-Driven Actions

Some actions are triggered by events, not directly by principals. When these actions are triggered, we need to find the capabilities of the responsible principals. Some events are statically registered via HTML code; for them, it is quite easy to find their principals by identifying the DOM nodes they belong to. Many other events are however registered dynamically, during the runtime, such as timer events, AJAX callback events, and those dynamically registered to DOM objects by Javascript code. Since the principals can be easily identified during registration, Contego binds the capabilities of the principals to the event handlers when the handlers are registered to events. Therefore, when these events are triggered, Contego can retrieve the capabilities from the event handlers.

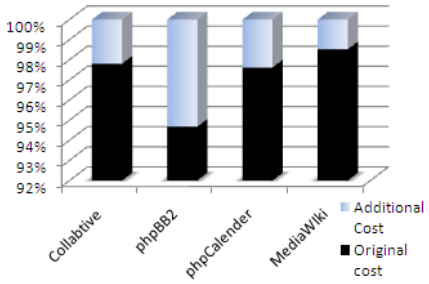

Fig. 2. Performance

3.5 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 measured how long it takes a page to be rendered in our modified browser versus in the original browser. We used 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 measured the total time spent on rendering pages and executing Javascript 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 2.

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

4 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. The full details of this work can be found in the extended version of the paper [9].

References

1. D. Crockford. ADSafe. `http://www.adsafe.org`.
2. M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities inweb applications. In *Proceedings of the Eighteenth Usenix Security Symposium (Usenix Security)*, Montreal, Canada, 2009.
3. 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.
4. C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW 2006*.
5. 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.
6. T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW 2007*.
7. C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *CCS 2007*.
8. B. Livshits and U. Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *PLAS 2007*.
9. T. Luo and W. Du. Contego: Capability-based access control for web browsers (full version). `http://www.cis.syr.edu/~wedu/Research/paper/contego_full.pdf`.
10. 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.
11. 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.
12. B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.
13. WhiteHat Security. Whitehat website security statistic report, 10th edition, 2010.