5-30-2012

A Data Flow Tracker and Reference Monitor for WebKit and JavaScriptCore

Andrew Bloomgarden
Dartmouth College

A Data Flow Tracker and Reference Monitor for WebKit and JavaScriptCore

Andrew Bloomgarden
Computer Science Department
Dartmouth College
Hanover, NH

Advised by Sean W. Smith and Sergey Bratus

May 30, 2012

Abstract

Browser security revolves around the same-origin policy, but it does not defend against all attacks as evidenced by the prevalence of cross-site scripting attacks. Rather than solve that attack in particular, I have opted for a more general solution. I have modified WebKit to allow data flow tracking via labels and to allow security-sensitive operations to be allowed or denied from JavaScript.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Web development is an often-frustrating field to be in. Oftentimes, features and experiences that would be easy to develop in native code using operating system frameworks are difficult to impossible to develop because of the restrictions of the browser environment. That might be an acceptable trade-off were security enforceable. While browsers are indeed a fairly secure runtime environment in terms of total system compromise, their notion of in-browser security is notoriously strait-jacketed yet full of holes. In this thesis, I describe how I modified WebKit and JavaScriptCore to allow web developers to encode their expectations of data flow into their sites.

WebKit is one of the most popular, standards-compliant, and fast-progressing browser engines today. It was forked from KDE's KHTML engine by Apple in order to build Safari [20] and has been adopted by other browsers over the years. Currently, it is used in Google's Chrome and Safari (both on the Mac and in iOS), as well as a number of less-well-known browsers. Chrome uses Google's V8 as its JavaScript engine, but most other WebKit browsers use the packaged JavaScriptCore engine.

In Chapter 2, I will describe the problem I am attempting to solve. In Chapter 3, I will take a step back and describe in some detail the existing browser security mechanisms. Others have attempted to solve data flow tracking in the browser before, so I will discuss their work in Chapter 4. Then, in Chapters 5 and 6, I will explain my modifications to JavaScriptCore and WebCore (WebKit's rendering engine) respectively. Finally, in Chapter 7, I will discuss how effective my changes are.

Chapter 2

The problem

2.1 The anatomy of a page load

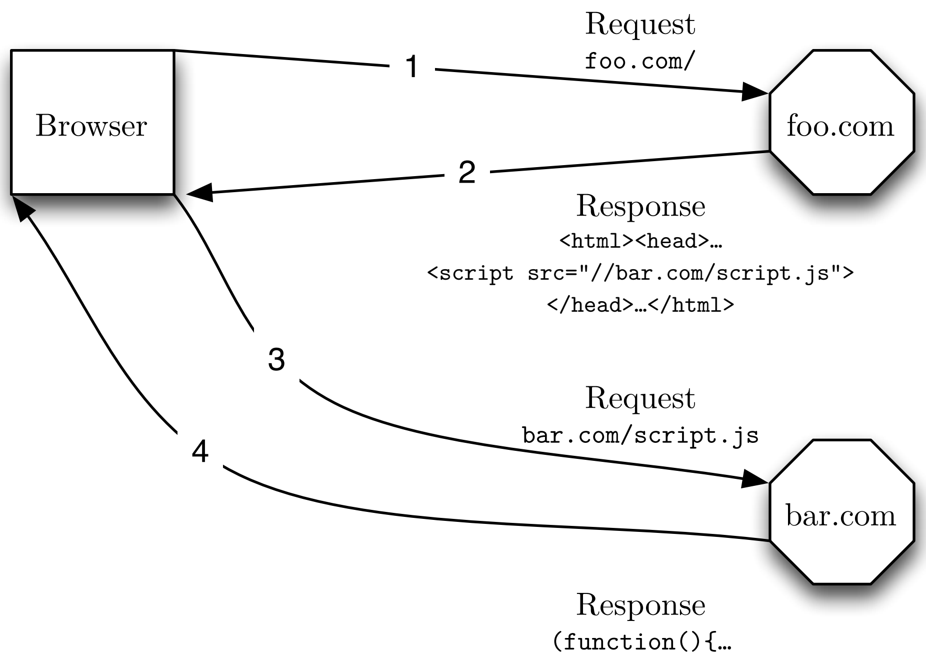
When a user browses to a web site, the following sequence of events happen:

- the browser sends a request to the server that hosts the site.
- the server responds with an html document.
- as the browser receives bytes from the server, it starts loading external resources as specified in the document from whatever server hosts the resource (see Figure 2.1). If the resource is a script, the browser executes the script immediately in the context of the page.

That last step should worry a security researcher, since it adds more points of failure to keeping the page secure. Obviously, if the initial server has been compromised, there is no hope for a secure system, but having external servers be able to provide libraries for the page forces us to consider them as possibly malevolent systems. Depending on machines beyond the organization's security perimeter for security makes the notion of a security perimeter much less useful.

Modern web development is often heavily reliant on externally-hosted scripts for essential functionality, whether from using Google's hosted jQuery or from using an API script hosted by a vendor. While there is nothing stopping a security-conscious developer from using a locally-controlled copy of jQuery, developers often have no choice when they need to make use of third-party services. External vendors often require that scripts be loaded from the vendor's domain in order to make it easy for them to push out updates. This is not an unreasonable requirement, because the promise of web applications is precisely that the service provider doesn't need to push out updates to every single user, but rather

Figure 2.1 Loading a page with an external script



just to their own servers.

However, the consequence is that those scripts are then not auditable by their users (i.e. the developers of a site using them) or by officials in charge of verifying a system's security. That either precludes their usage in truly security-conscious settings or causes the developer to simply cross their fingers that nothing will go wrong.

2.2 The consequences of a rogue script

Here is a concrete example of the consequences of a developer sacrificing security for utility. Imagine that Alice is building a website for a newspaper. She needs to add Facebook's "Like" button to the article page, but she doesn't want it to be able to read her site's cookies in Bob's browser and send them off to Facebook. It should, however, be able to see the current page's URL, because that's the only way that the newspaper's page can be liked. Because the Facebook JavaScript SDK is implemented as a script hosted by Facebook, the page has to include a `<script>` tag to instruct Bob's browser to fetch it from Facebook. That means that Alice can't audit it with any confidence, since the file she audits at development time could be quite different from the files the newspaper's users see next month, from a different location, or from any of other various factors.

Figure 2.2 A function and its minified version from jQuery

```
// unminified
noConflict: function( deep ) {
  if ( window.$ === jQuery ) {
    window.$ = _$;
  }

  if ( deep && window.jQuery === jQuery ) {
    window.jQuery = _jQuery;
  }

  return jQuery;
}

// minified
noConflict:function(b){a.$===e&&(a.$=g),
  b&&a.jQuery===e&&(a.jQuery=f);return e}
```

Not only is there the difficulty of not controlling changes to a script, it can also be difficult to understand what the script might do in the first place because of *minification*. Minification is automated rewriting of JavaScript to remove excess whitespace, and more importantly to shorten it as much as possible using whatever tricks may exist. This includes rewriting variable names, stripping out comments, and reordering files where it would not make a difference. To see how difficult this can make auditing, Figure 2.2 is a simple function from jQuery, one of the most common JavaScript libraries on the internet, followed by the same chunk of code from the minified jQuery [22].

Even though it's computationally simple to bring back the whitespace, it's clear that a lot of information useful to humans has been lost, from variable names to code ordering. Just imagine trying to decipher a full library that had been minified—it's not fun. More importantly, it's likely that Alice will miss something important.

2.3 Previous attempts to solve the problem

Nevertheless, Alice's request is simple: don't let an external script steal sensitive data that it shouldn't be able to from Bob's browser. I will attempt to solve this problem, but not at the expense of a more general solution.

This class of problem can't necessarily be solved by hard-coded tools that prevent broad classes of actions on sensitive data, as they could easily break necessary functionality. For example, Vogt et al. [26] have built a taint-based tool to prevent any data like the page's URL from being transmitted to a non-origin server without the user's approval. It's not a particularly nice user experience to get pop-ups requesting permissions, nor is it likely to make users pay attention to what is happening (think about how people ignored Windows Vista's ever-present User Account Control prompts).

Tools like BFlow [29], on the other hand, do accomplish much of this goal, but require a broad restructuring of a document's structure and additions to the site's server side. BFlow does implement a data-flow-based security system for the browser, but it hard-codes the policy it enforces into the browser. Moreover, it only enforces mediation at the granularity of a frame, but most sites are not structured as many independent frame-modules.

2.4 My approach

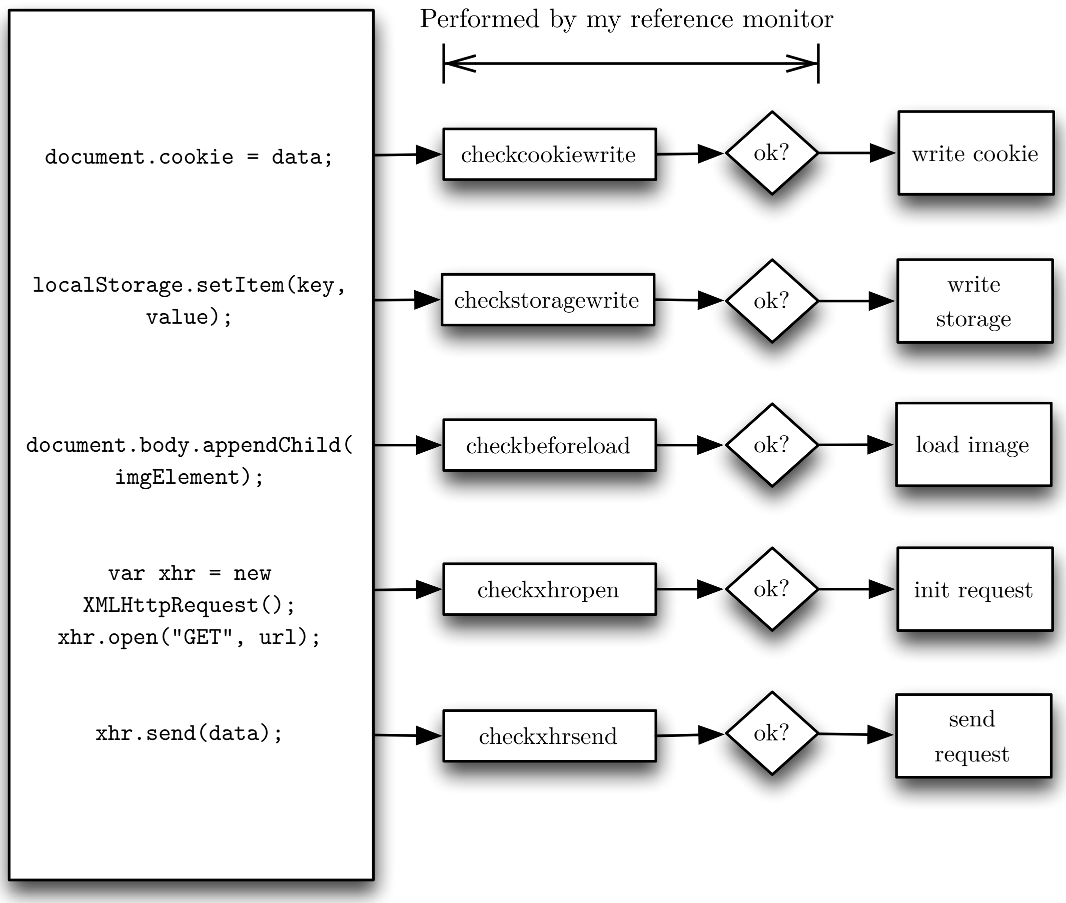
In order to allow developers to codify their intent in this case and in others, I have built a new reference monitor into WebKit to allow developers and researchers to instrument code and data. A classical concept in computer science ([17, 25]), a reference monitor is the combination of a security policy and a series of hooks that can be utilized to enforce that policy. While browsers do include a reference monitor in the form of the same-origin policy, which prohibits whole classes of dangerous and benign actions, the only thing it makes decisions based on is the current frame and its domain. Mine lets the developer define policy based on data flow, allowing the developer to both solve this problem and possibly others.

Most web applications are built with sensitive data stored side-by-side with non-sensitive data, so what I have done is to implement a reference monitor for the browser that works at object-level granularity. A default policy can be specified in under 500 bytes of JavaScript code, allowing it to easily be inserted at the beginning of the page, but the developer can choose to enforce any policy she chooses. All that matters is that the policy enforcement code be included before any other JavaScript, which is a fairly trivial requirement. This system gives the power to the developer, who is best capable of judging what should be allowed or not.

I implemented a label-propagation system in JavaScriptCore, the JavaScript implementation used in Safari and in most non-Chrome WebKit-based browsers, that allows the developer to arbitrarily label values and have those labels be propagated to the result of any operation that uses a labeled value as an input. I also selected a few DOM properties that need to have a known label applied on any site, although I do not apply any default policy for those labels.

Finally, in order to be accessible to web developers, the enforcement side of the system is built using events that the developer can bind callbacks to. Just like attaching a function to a `click` event, the browser allows the developer to attach callbacks to each of the following event types (see Figure 2.3):

Figure 2.3 Some of the callbacks, illustrated



- writing data to cookies
- writing data to `localstorage`, a client-side persistent key-value store
- loading any resource, whether image, script, or CSS
- making an `XMLHttpRequest` (the X in AJAX)
- copying data to the system clipboard

On each of those events, the object in question is passed to each function, which can examine the labels it possesses. If any function returns false, permission is denied. This can be used by loaded scripts to deny all mediated operations, but the consequences of that do not lead to privacy or security violations, just denial of service.

To show the potential of the reference monitor to make site security easier to reason about, I implemented an example page that has different types of sensitive data on it:

- a token to prevent cross-site request forgery
- the cookie stored by the browser
- the user's name

Like many sites, this page loads JavaScript from a domain other than its own into its own frame, giving the script full privileges. I've implemented a policy for this page using the reference monitor that prevents any request from being made containing sensitive data that was not made from a specific, permissive context. This is an example of encapsulating the developer's intent: the developer expects that requests are going to be made with this sensitive data from a specific place and nowhere else.

What is most important about this reference monitor is that it is much more problem-agnostic than prior work. Instead of focusing on cross-site scripting or data leakage and hard-coding a solution into the browser, a reference monitor allows for a variety of uses.

For example, it could be used to make up for a lack of trust in vendors as well as to stop cross-site scripting. Cross-site scripting in ways an easier problem to solve than untrusted vendors, because complete sandboxing mechanisms can guard against script execution where user input can affect the page. However, vendor scripts need to be executable and need to phone home, so my work can help ensure that they don't send home what they are not supposed to.

It could also be used by a developer to enforce application-internal policy, would then be made much more obvious, for example if the reference monitor alerted the developer that something had gone horribly wrong. If this technology were incorporated into browsers, researchers would be able to stop re-inventing the wheel by adding data flow tracking to browsers and simply solve other problems. In turn, developers could do more than cross their fingers that the libraries they loaded would not turn malicious or unintentionally transmit sensitive data.

Chapter 3

The existing browser reference monitors

When I say that I have added a reference monitor to WebKit, it's not because the browser is a complete and total free-for-all without any security restrictions. In fact, the situation is quite the opposite: there is a reference monitor in any standards-compliant browser already, but it has many problems.

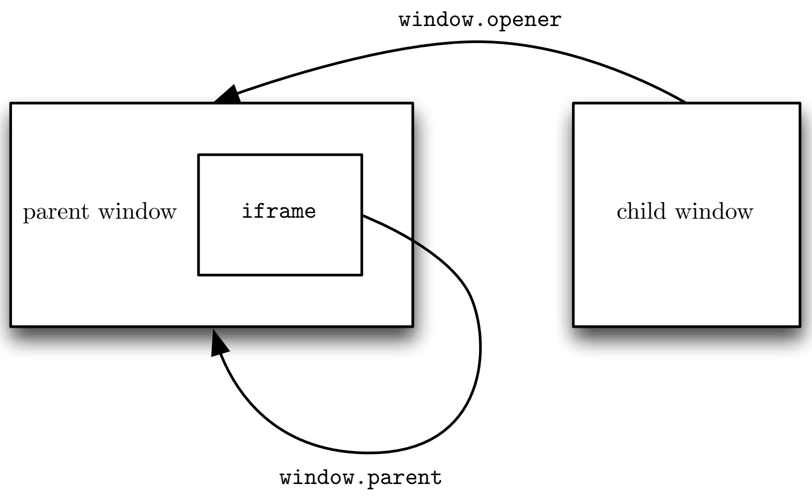
Reference monitors can be thought of as managing information flow between different contexts. In a traditional operating-system-level reference monitor, a context can be anything including processes, sockets, and files, but contexts can be anything. However, whatever the definition of a context is, what is eseential is that the more granular the contexts, the finer-grained the policies that can be enforced by the reference monitor [17].

The existing browser reference monitor enforces what is called the *same-origin policy*, which attempts to prevent data leakage and malicious interaction between content from different servers. The same-origin policy is quite strict, but its notion of context is solely based on a frame's domain [30]. Unix permissions have a somewhat coarse-grained notion of context, consisting of users, groups, and files, but the same-origin policy is even coarser-grained than that. To show the same-origin policy's shortcomings, I will walk through where it is enforced, first in the DOM and then in `XMLHttpRequest` creation.

3.1 DOM

The DOM, or Document Object Model, is the standardized in-memory interface to the browser's understanding of the current document [16]. It is a many-child tree structure, with many different types of nodes. JavaScript code is able to walk and query the tree,

Figure 3.1 Child windows and their references to their parent



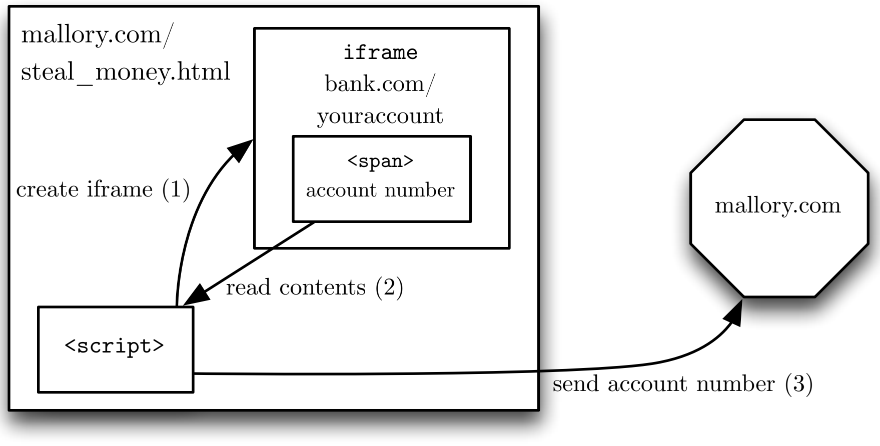
create and delete nodes, and modify the contents and properties of nodes, provided that the specification allows it. Additionally, and quite importantly for any sort of interactivity, JavaScript can bind callbacks to events specified by the HTML specification [11], such as clicks, mouse movement, text entry, resource loads, etc. From a security perspective, then, JavaScript has the power to read data from the page and the power to do something with it.

One of those classes of nodes in an HTML document can be the parent of entire subdocuments, as shown in Figure 3.1. The `iframe` element, for example, can load documents from across domains and can be created in the initial markup or by JavaScript. New windows can be created to access cross-domain pages, but they are still linked to their parent window and can access them via `window.opener`.

Here is the first instance of a major security problem were the same-origin policy not enforced. When a browser loads a resource from domain `foo.com`, the browser sends along any cookie data that it has stored for that domain. Because HTTP connections are stateless and because HTTP allows servers to instruct user agents to store cookie data on their behalf, cookies are the standard way for sessions to be maintained in browser sessions [2]. As shown in Figure 3.2, a malicious page could then easily steal all sorts of information from sites Bob is currently logged into by creating `iframe`s targeting known URLs, reading sensitive data off of the result, and then transmitting it back home. Similarly, if a malicious document were loaded into a subframe, it could steal sensitive data from the parent.

Therefore, the same-origin policy prevents documents from reading or writing into subtrees or into parents across domains. A frame can only access another if its fully-qualified domain name, protocol, and port are the same as the target's. There is a small way out for frames from different domains wanting to access each other. If two frames with

Figure 3.2 Stealing data without the same-origin policy



domains a and b share a common super-domain or if a is a super-domain of b or vice versa, they can set `document.domain` to that super-domain [30]. This must be done by mutual agreement between frames. This is enforced by clearing the port when `domain` is manually set. A document from `http://foo.bar.com` cannot communicate with `http://bar.com` by setting its `document.domain` to `bar.com`, because while the first frame's context is `bar.com`, the second's is still `bar.com:80`.

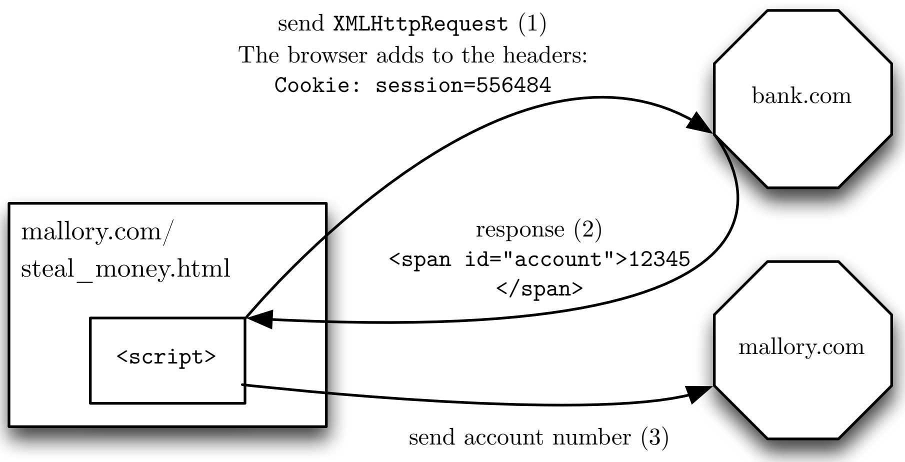
Zalewski notes that, if different documents are able to cooperate this way, setting a mutually-agreed-upon `domain` property allows any sub-domain of that property to also read those documents' data [30]. A parent domain of user-controlled subdomains, then, would be advised not to use this technique, or any of those subdomains could read data from the parent domain without restriction.

3.2 `XMLHttpRequest`

One of the big buzzwords in web development has been AJAX (Asynchronous JavaScript and XML), reflecting the shift from developing sites to applications. AJAX depends on the `XMLHttpRequest` object, originally implemented on Microsoft in Internet Explorer but since implemented in all browsers and standardized by the W3C [15]. `XMLHttpRequest` implements a simple API, allowing JavaScript to schedule an HTTP request to a remote server and asynchronously be given the response.

Again, though, there's an obvious security hole. Any request made by an XHR will pass along any cookies in browser storage, so if requests were unrestricted, a visit to `mallory.com` could read Bob's bank account numbers by simply requesting pages from

Figure 3.3 Stealing data without the same-origin policy using an `XMLHttpRequest`



Bob's bank and then sending it back home (shown in Figure 3.3). All of this could happen with Bob being none the wiser.

To stop this in its tracks, the standard specifies that every XHR request will be subject to the same-origin policy. Therefore, `http://mallory.com` could not make a request to any site other than `http://mallory.com`. This, of course, is highly restrictive, making legitimate cross-site development extremely difficult.

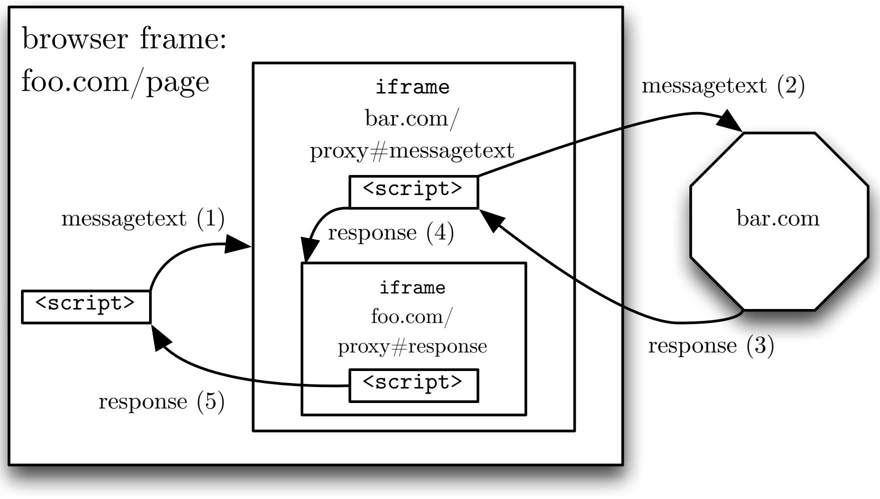
3.3 Breaking the same-origin policy

Consider the example of the Facebook JavaScript SDK that I mentioned in the introduction. It's most often used by page developers to put a "Like" button on the page. The desired behavior is simply this: the user clicks on the button and Facebook records the Like. Redirects are out of the question. There should be no sign to the user that there is anything awkward involved in this interaction, so pop-ups are not a good solution, even if they're opened in the background.

If this were code that could use standard OS APIs, there would be no issue making a request. A native Facebook SDK would just make an HTTP request with the appropriate parameters and interpret the results as it pleased. This is clearly what the developer wants, because she included it in her application and called it.

However, as I noted above, in a browser this isn't so simple. The developer could add an `iframe` with the `src` attribute set to the request URL, but the developer couldn't examine

Figure 3.4 Cross-domain communication using `iframe`s



Cross-domain communication requires two `iframe`s, one from the parent's domain and one from the target's. The parent sets the URL hash to the message body and triggers an event to notify the child, which in turn sends an `XMLHttpRequest` to the target server. It then lets its child frame know about the result, which can then notify the top-level frame via `window.top`.

the response. On a pedantic level, the request would be made as a GET, which is assumed by the HTTP 1.1 RFC to be idempotent [8]. Therefore, this is inappropriate for requests that change some state in the recipient. The same holds with inserting an `img` element.

Therefore, developers have come up with a fairly elegant hack called "fragment identifier messaging" [5]. It subverts the same-origin policy, although only with the target domain's cooperation. A script wanting to communicate with another domain loads a message-sending page from the target domain in an `iframe`. Then, when it wants to send a message, it sets the `iframe`'s URL's location hash (the part after the #) to the encoded message. Then, it notifies the frame, often via a resize event, so the frame can read the event and take action. A response can then be delivered by modifying the parent's location hash, as illustrated in Figure 3.4. Extensions of this core idea that use two proxy frames, one from each domain, to communicate between two full frames from different domains [1].

This whole procedure is quite inelegant and only necessary because of a security policy that, while doing its job, gets in the way too often. This `iframe` communication mechanism has been standardized in the form of the `postMessage` API [11]. All it does is provide an approved way of sending messages, but it's still an odd way of allowing communication.

What this boils down to, though, is that when Alice loads a script from an external domain, there is nothing stopping it from reading data off of her page and shipping it off to a cooperating domain.

3.4 The makings of another reference monitor

3.4.1 The tools

Following the same-origin policy clearly requires the browser to perform checks whenever something that might be prohibited could happen. However the browser developers choose to implement it, they are clearly implementing a reference monitor, albeit one with an extremely limited notion of context and label.

However, there are also the makings of another reference monitor in the browser, one directly controllable by the developer: the ability to prevent the default actions of certain events. For example, suppose Alice were working on a registration form for a site. It's quite important that Bob provides an email address when registering, but she doesn't want to have the only validation be on the server, returning a new copy of the form with the error marked. There's a simple solution: use the DOM event API along with HTML events to validate the input, as shown in Figure 3.5.

There are some other events that have associated default actions, most notably the `click` event. These events are often used to provide added behaviors and not strictly to enforce policies, but they can certainly be used as part of a reference monitor.

3.4.2 The restrictions

Because event handlers that execute arbitrary Turing-complete algorithms can be attached to arbitrary elements, the class of policies they can enforce or transformations they can execute are infinite. However, there are policies that it can not enforce, simply because the hooks that are available are not sufficient. Specifically, there aren't standard events that can be used to examine and block resource requests. WebKit does have a `beforeload` event that can will fire on all resources, but it does not always prevent the default when asked. Also, events do not fire across documents, even when nested, so hiding a prohibited operation in a different frame is an effective way to prevent mediation by a reference

Figure 3.5 Adding an event listener

```
formElement.addEventListener('submit', function (e) {
  if (!checkForm())
    e.preventDefault();
});
```

monitor [11].

One more constraint is that, without tracking data flow, there is little that can be done to protect sensitive data. Unlike an operating system, where the reference monitor often makes decisions based on fairly fine-grained contexts, the notion of context in the browser is still quite limited. Without that fine-grained notion of context, an operating-system-level reference monitor would be forced to deny network access based on very limited information in order to prevent data leakage. That's precisely the issue that I'm trying to solve in browser security.

To improve security, either the notion of and introspection into context or data flow must change. Improving context separation would require serious efforts on the part of developers, since it would a major change in how web applications load themselves in the browser. Tracking data flow, though, would require much less work on the part of the developer, so I think it's much more likely to be adopted. Therefore, I focused my efforts on tracking data flow.

Chapter 4

Prior work

4.1 Netscape

There has been a significant amount of research done to solve the problem of how to provide some reasonable information about data flow in JavaScript. The most common solution has been to attempt to modify the JavaScript interpreter to associate taint with objects and values, allowing some policy to be enforced.

In fact, Netscape itself originally attempted to solve this problem in Navigator 3.0. They implemented data tainting in JavaScript, where "taint" here is not a single bit, but rather an accumulation of data origins [3]. If Alice wanted to taint a value, she could execute

```
var taintedValue = taint(value);
```

which would add the current script's origin to the accumulated taint. A script could similarly remove its own origin from a value by calling `untaint`. The browser would then prevent the transfer of tainted data to a server in any case where the transmitted data was tainted with a different server's origin.

Unfortunately for contemporary web developers, this system was not long for this world. Navigator 2.02 had introduced the same-origin policy, but clearly Netscape thought that it was too restrictive if they went through the effort of developing an alternative system for 3.0. However, to turn on this data-tainting feature, the browser had to be launched with an environment variable set, making it much less likely to be adopted by developers.

With JavaScript 1.2 [4], released with Navigator 4.0, tainting was removed in favor of signed scripts as the new direction of browser security. The developer could assert that the script was unmodified by providing a Java JAR file with signatures and requests for privileges for when the script matches the signature. The same-origin policy was left in for unsigned scripts, but signed scripts were able to request the ability to violate it.

Although that kind of blanket privilege would be quite problematic today, the ability to enforce a signature for a script that a page was loading would be extremely useful. Signed scripts never really caught on, unfortunately, so the same-origin policy was left as the standard.

4.2 Academic research

Clearly, though, the same-origin policy is not sufficient for a truly secure browser, nor is it ideal for developers. As years have passed, new classes of vulnerabilities have appeared. Cross-site scripting attacks are to web development as buffer overflows are to system development, in that it's really easy to introduce a vulnerability unless the developer is consistently following best practices. Unlike buffer overflows, however, where a successful exploit gives control of a system, a cross-site scripting exploit can only give control over the frame it was loaded into—which is to say, not too much control. The only real danger is in sensitive data being stolen, so researchers have focused their efforts on data-leakage prevention.

In their TaintSNIFFER paper [21], Miller and Sandhu built a single-bit tainting system into Microsoft Research's C3 browser-research platform [18]. While it does specify a reasonable policy for propagating taint, as well as providing a decent test suite, there is no policy or mechanism given or implemented for using taint to do anything. One notable missing feature is that their implementation requires the use of C3's JavaScript interpreter, missing out on its just-in-time (JIT) compiler's performance benefits.

Alhambra [24], developed by Tang et al., also uses taint-tracking, this time to enforce hard-coded policies in a WebKit-based browser. They claim no measurable overhead for taint-propagation. I'm unclear how there is no overhead, unless they are only using the interpreter and not WebKit's faster JIT.

Vogt et al. built a taint-tracking engine [26] that has one major benefit: it tracks implicit data flow by both tainting any results from tainted control flow decisions and by performing static analysis to prevent other leakage. However, again, their only goal was to prevent cross-site scripting, so they hard-coded in a policy.

None of the above systems try to prevent data from moving between in-memory contexts. Because JavaScript works almost entirely with in-memory objects, structuring a system so that a mandatory access control system could be implemented to decide whether data could be moved between contexts would be difficult.

Yip et al. do implement such a system in BFlow, but their notion of context is unfortunately the frame, like the same-origin policy for JavaScript manipulations. Labels are only kept track of on each frame, so in order to construct a reasonable system, a site must be broken up into discrete frames to have any policy-checking. There is also a server-side component to specify what labels data carries, further complicating the implementation for developers. The system, though, does provide the useful notion of tags as objects created by developers in order to mark different kinds of sensitive data.

Chapter 5

My work on JavaScriptCore

5.1 Labeling

One major goal of my modifications to JavaScriptCore is to ensure that I am not hard-coding a policy into place or overly restricting the policies that can be enforced. Single-bit tainting is useful when implementing hardware-level data protection, but it is not too useful when policies need to coexist. Dalton, Kannan, and Kozyrakis think that only four policies are necessary in their Raksha dynamic information flock tracking system, so they provide a four-bit tag structure [6].

I don't think that four bits is enough to let arbitrary code enforce arbitrary policies. Specifically, when one piece of code is trying to track a specific piece of data through the system, I don't want another piece of code to be able to tamper with the flow tracking without the first piece of code's permission.

Therefore, I use a system similar to an operating-system-level labeling system, Flume [10]. Every value can have a label, which is in turn a set of tags. Each tag is opaque and managed by the operating system, which in this case is the JavaScript virtual machine. In my implementation, tag objects are the only interface onto a value's security label. Without a tag reference, labels are completely invisible.

5.2 The API

To implement the above notion of tags and labels, I implemented the highly-constrained API shown in Figure 5.1.

`SecurityTag`s are opaque objects that can only add themselves to values or check their

Figure 5.1 The `SecurityTag` API

Method	Description
`tag.addTo(object)`	Adds `tag` to `object`, returning the tagged version.
`tag.isOn(object)`	Returns `true` if and only if `tag` is in `object`'s label.

Figure 5.2 Tagging primitives and objects

```
var tag1 = new SecurityTag(), tag2 = new SecurityTag();
var labeledBy1 = tag1.addTo(1);
var labeledBy2 = tag2.addTo(1);
var labeledByBoth = tag1.addTo(tag2.addTo(1));

print(tag1.isOn(labeledBy2)); // false
print(tag2.isOn(labeledBy2)); // true
print(tag1.isOn(labeledByBoth)); // true

var object = {};
tag1.addTo(object);
tag2.addTo(object);

print(tag1.isOn(object)); // true
print(tag2.isOn(object)); // true
```

own presence. Without a reference to a specific tag object, it is impossible to reason at all about labels. Because JavaScript has closures that cannot be peered into, it is quite easy to keep references secret.

The semantics of adding a tag to a value is different depending on whether or not the value is a primitive (Figure 5.2). If it is, the argument to `addTo` is unchanged, while the return value has the tag added to it. For objects, the parameter is changed. This behavior corresponds to argument-passing behavior, in that primitives are passed-by-value and objects are passed-by-reference.

I also marked the `SecurityTag` constructor and prototype as frozen, as well as making its reference in the global object not `[[Writable]]` and not `[[Configurable]]`. Otherwise, the entire system could by undone by simply overwriting the key functions with less honest ones. The `SecurityTag` symbol can be shadowed, but because it's trivial to get access to the global object even when it's out of scope, the developer can refer directly to the global object's `SecurityTag` property to be safe.

5.3 Labeling data

When building a data-labeling system, one of the most important decisions to be made is where to store the label. Tagging systems that can run on uninstrumented binaries often

Figure 5.3 The distinction between objects and primitives

```
var booleanPrimitive = false, booleanObject = new Boolean(false);
var resultFromPrimitive, resultFromObject;

if (booleanPrimitive)
  resultFromPrimitive = "truth-y";
else
  resultFromPrimitive = "false-y";

if (booleanObject)
  resultFromObject = "truth-y";
else
  resultFromObject = "false-y";

assert(resultFromPrimitive == "false-y");
assert(resultFromObject == "truth-y");
```

This figure shows that, while boolean primitives evaluate in conditionals as expected, `Boolean` objects always evaluate as true.

build a single static data structure to store tags, indexed by memory address [31] [13]. Hardware implementations extend registers and memory to store tag data right alongside the labeled data [6].

Object-based labeling schemes are built similarly to hardware implementations, except that they need to find a space in the object representation to store labels. SELinux, for example, modifies kernel data structures to include a `void*` field that can point to security data [23].

JavaScript would seem to lend itself to such a simple modification, but unfortunately the language is not as simple as many would like it to be. When Brendan Eich, the creator of JavaScript, was defining the language, his bosses were hoping to make it similar to Java in order to ease adoption [7]. While the syntax is most clearly C- and Java-like, one other major component he brought over was the distinction between primitive types and objects. This distinction is not always the most intuitive or seemingly correct, as shown in Figure 5.3.

An object always evaluates to true when the internal ECMAScript method `ToBoolean` is called on it [12]. My original solution when labeling values was to auto-box anything that needed a label, but that quickly fell apart when those labeled values were used under the assumption that they were primitive. Therefore, I needed to label objects and primitives separately.

5.4 JavaScriptCore value representation

In JavaScriptCore, all values can be passed around as `JSValue`s. Those values can be either immediates (non-string primitives) or pointers to `JSCell`s. The exact representation of those primitives is different depending on whether or not the compiler is told to generate code for a 32-bit or 64-bit processor, but in either case a `JSValue` is only 64 bits in size.

For an arbitrarily-sized label, that simply is not enough space. There has to be a pointer to the label object, but consider a `JSValue` holding a double-precision floating-point number (a double). That takes 64 bits of space, leaving no room whatsoever for a pointer.

Thankfully, the JavaScriptCore architecture gave me a reasonable solution. JavaScript objects are stored as `JSObject`s, and primitive strings are stored as `JSString`s. Both of those classes inherit from `JSCell`, so I added a label field to `JSCell`. That let me easily label strings and objects, but there was still the issue of other primitive types.

I then created `JSLabeledValue`, a new subclass of `JSCell` that simply holds a `JSValue`. Instances of that subclass can then be labeled. To label an immediate, then, I allocate a new `JSLabeledValue`, store the value to be labeled in it, store the pointer to the label, and finally return the pointer to the labeled value.

The one hiccup in this implementation is that operations that look at `JSValue`s must be modified to realize that labeled values should be treated the same way as their unlabeled counterparts. Otherwise, the benefits of labeling primitives without auto-boxing them to objects is missed.

5.5 Label representation

Internally, the `SecurityLabel` API is also quite simple. Any object can have a label by having a `SecurityLabel` field. That class provides the API shown in Figure 5.4.

Figure 5.4 The SecurityLabel API

```
public:
  bool isNull() const;
  void add(const SecurityTag& tag);
  bool hasTag(const SecurityTag& tag) const;
  void merge(const SecurityLabel& other);

private:
  RefPtr<SecurityLabelImpl> m_impl;
```

While this implementation is not the most efficient, it does attempt to share the actual label values. It holds a reference-counting pointer to the actual label implementation, allowing it to initialize itself off an existing label easily. It can also safely merge in another label while making sure not to accidentally label any other objects pointing to its current `SecurityLabelImpl` instance. This additionally allows `SecurityLabel`s to be pass-by-value, since they're only the size of a pointer, letting consumers of the class pay no attention to the memory management of the label.

A `SecurityLabelImpl` is currently implemented as a `HashSet`, a WebKit class implementing an $O(1)$-time set using a hash table. The `SecurityTag`s are just `double`s, initialized using WebKit's Web Template Framework's `monotonicallyIncreasingTime` method. This is just an artifact of the current implementation, since it would be trivial to change the definition of a tag. All that is required is that they are different depending on when they were initialized.

5.6 Propagating labels

The TaintSNIFFER paper [21] provides a reasonably comprehensive test suite for single-bit taint propagation, so my first order of business was modifying JavaScriptCore to pass the suite with taint replaced by a single-tag label. Unfortunately, like C3, there are multiple execution engines in JavaScriptCore, an artifact of its development and its many targets. They are as follows:

- The "classic interpreter". [1]
- The just-in-time compiler (JIT).
- The data-flow graph JIT (DFG JIT) [28].
- And recently, the low-level interpreter (LLINT).

All of these implement the same bytecode semantics, but are enabled depending on both compile-time and run-time characteristics. Conveniently enough, if the classic interpreter is enabled in a build, the JIT, DFG JIT, and the LLINT are disabled, so it was a good place to start. Additionally, it is the most simple of the interpreters, as it is just a `switch`- or `goto`-based engine implemented only in `C++`. It did not take much time to patch it to propagate taint.

[1] I've put the classic interpreter in quotes because of its origins and as a cautionary note. It was only a few years ago that it was the new bytecode-based replacement for the old AST-based interpreter [28]. However, since then, there have been three other execution engines introduced, all of which have separate implementations for most opcodes. Truly supporting data labeling requires both modifying all existing engines and keeping up with the rapid pace of development.

Unfortunately, all three of the other interpreters are assembly-based. The JITs generate it at run-time, while the LLINT generates it at compile-time in order to be a faster version of the classic interpreter. Of course, because these generate assembly, they are have components implemented both for 32-bit processors and for 64-bit processors, increasing the number of components that I needed to change by a factor of two.

Thankfully, although these engines were implemented to be faster than the classic interpreter, they still often call `C++` functions when encountering `JSValue`s pointing to `JSCell`s. In other words, their optimizations are mostly for immediates, not strings or objects. Therefore, many of the changes I needed to make were in `C++`-land. For example, for every engine, I needed to modify the slow path implementation for `op_not`.

However, that is a good example of some changes I had to make to assumptions the engines reasonably made. In standard JavaScript, the result of a `not` operation is an immediate boolean, but with my changes it can be either an immediate boolean or a labeled boolean. One is represented as a pointer and the other is not, so I had to force it to consider that possibility. Similarly, the result of subtraction, multiplication, and division are always numbers in standard JavaScript, but obviously if label propagation is working, the result may not be a number.

After many such changes, the TaintSNIFFER test suite was passing under all engines. I also, however, had to develop a test suite for JavaScript functions implemented in native code, because any changes to bytecode processing don't do a thing for `C++` code. I modified all of them to get the test suite to pass.

Chapter 6

Instrumenting the browser

6.1 Propagating labels in the browser

Adding label propagation to the WebCore, the DOM implementation for WebKit, is much simpler in theory than adding it to a JavaScript engine because of the value types involved.

In WebCore, much more than in JavaScript, values are strings referenced by objects in tree structures. Any propagation, then, just needs to handle operations on strings and conversions between strings and nodes. WebCore uses the same underlying string class for almost all its operations, `WTF::String`, so it was trivial to add a new field in that class to store labels. Conveniently enough, like my `SecurityLabel` class, a `String` is just a reference-counted `StringImpl`, so labeling one does not immediately label all references to the same string. Handling string operations is then trivial, because they are nicely encapsulated. DOM nodes are all descended from `WebCore::Node`, so again adding a label field is easy enough.

That leaves handling conversions between strings and nodes/trees. The DOM API lets developers read and write string descriptions of node structures in the form of HTML, allowing developers to create whole new sections of documents by writing what they are familiar with from JavaScript. To handle writing `innerHTML` (parsing HTML), I had to patch WebCore to label all new nodes with the string's label. To handle reading `innerHTML`, I patched it to generate a string with the accumulation with the labels of all nodes and attributes. Handling writing `innerText` was easy, because it is only looking at a single node (if it is a text node) or creating a single node. Reading it is similar to `innerHTML`, in that it looks at the text of all children, so I patched it to similarly accumulate labels.

That handled operations and types inside WebCore, but that does not account for the actual conversions made to and from JavaScript types. Because WebKit can be linked against other JavaScript engines (most notably Google's V8), and because the specs define them that way, all WebCore objects manipulable from JavaScript are described using an interface definition language (IDL) similar to the WebIDL spec [19]. Those bindings are then read by a script that generates `C++` bindings, so I just needed to modify a couple of conversion functions and the bindings generation script to pass along labels.

6.2 The reference monitor

As I discussed in Chapter 3, there is already some infrastructure needed for implementing a reference monitor controllable from JavaScript. The events given by the specs, however, are lacking in a number of ways.

6.2.1 Decision-making data

First off, the events fired by the browser do not make it easy to determine if labeled data is going to be sent or written somewhere. The `submit` event, for example, fires on form submission, but it does not provide any easy way to access the data that is going to be sent or the URL it is going to be sent to. A policy enforcer would have to walk the entire form tree in order to check for the presence of labels.

It should be simple for decision-making code to see exactly what it needs to check. More importantly, because labels are opaque without references to tags, they are safe to send cross-frame. Thus, I created a new event class, `SecurityEvent`, which at the very least holds a `SecurityLabel`, as shown in Figure 6.1.

6.2.2 New events

Existing event code is expecting either simple events (events with no associated data) or other kinds of `Event` objects, so I needed to add a series of new events. Also, I needed to add some events because there were no usable hooks in place already.

Therefore, I added the events listed in Table 6.1. When fired, all of them are called with a `SecurityEvent` representing the accumulation of labels from all the data that is

Figure 6.1 Listening for a SecurityEvent

```
// tag is a SecurityTag
window.addEventListener('checkxhrsend', function(e) {
  if (tag.isOn(e.securityLabel))
    return false;
});
```

Event	Trigger
`checkbeforeload`	loading a resource (image, script, etc.)
`checkcookiewrite`	writing `document.cookie`
`checkcopy`	copying to the clipboard
`checkcut`	cutting to the clipboard
`checknavigate`	changing the location of a frame
`checkpostmessage`	calling `postMessage`
`checkstoragewrite`	writing something to DOM storage
`checkxhropen`	calling `open` on an `XMLHttpRequest`
`checkxhrsend`	calling `send` on an `XMLHttpRequest`

Table 6.1: New events for my reference monitor in WebCore

Figure 6.2 Labels do not bubble to the top of containers

```
var obj = {}, tag = new SecurityTag();
obj.labeled = tag.addTo("sekrit data");
print(tag.isOn(obj)); // prints false
print(tag.isOn(obj.labeled)); // prints true
```

going to leave memory.

For structured objects, the labels do not automatically bubble up to the top. For example, the code in Figure 6.2 will print false.

However, if the object is serialized for transmission over the network, the resulting string will have that tag on it. Therefore, the labels that I pass to the event handlers are done after ensuring that the data to be transmitted has been flattened into a string.

6.2.3 Attaching and dispatching

As discussed in Section 3.4, events do not bubble cross-frame. "Frame" in this case can be anything from an `iframe` nested inside another frame to an entire other window opened via `window.open`. Whether cross-origin or not, communication can happen between these frames. That means that data that should be protected by an event handler could leave its original frame, making it critical that that policy gets tested no matter what frame the policy was originally created in.

Worse, before anything that might invoke the reference monitor might occur, the original window or document could disappear. Because events are normally attached to the window, document, or child of the document, if those disappeared the events would not fire.

It should be clear that the normal method of attaching events will not work for a monitor that can provide anything approaching complete mediation. My solution is to not attach security event handlers directly to the window, but rather to a reference-counted object pointed to by that window. Crucially, a `DOMWindow` object that represents an `iframe` will look to the topmost window for the event holder, not itself, while a new window opened by `window.open` will point to its opener's event holder.

Adding a cross-frame event dispatcher is risky, though, because of the possible security consequences. An event handler from one frame should always run in the context of its original global object, because otherwise it would be able to bypass all same-origin policy restrictions. With judicious manipulation of JavaScriptCore's mark-and-sweep garbage collection visitation, I forced references to old `JSDOMWindow` objects to stick around past when their corresponding frames disappear. Then, when a cross-frame event fires, it always can run with its original global object, even if that object has been stripped of most of its power by being detached from its frame. I've attempted to illustrate this situation in Figure 6.3.

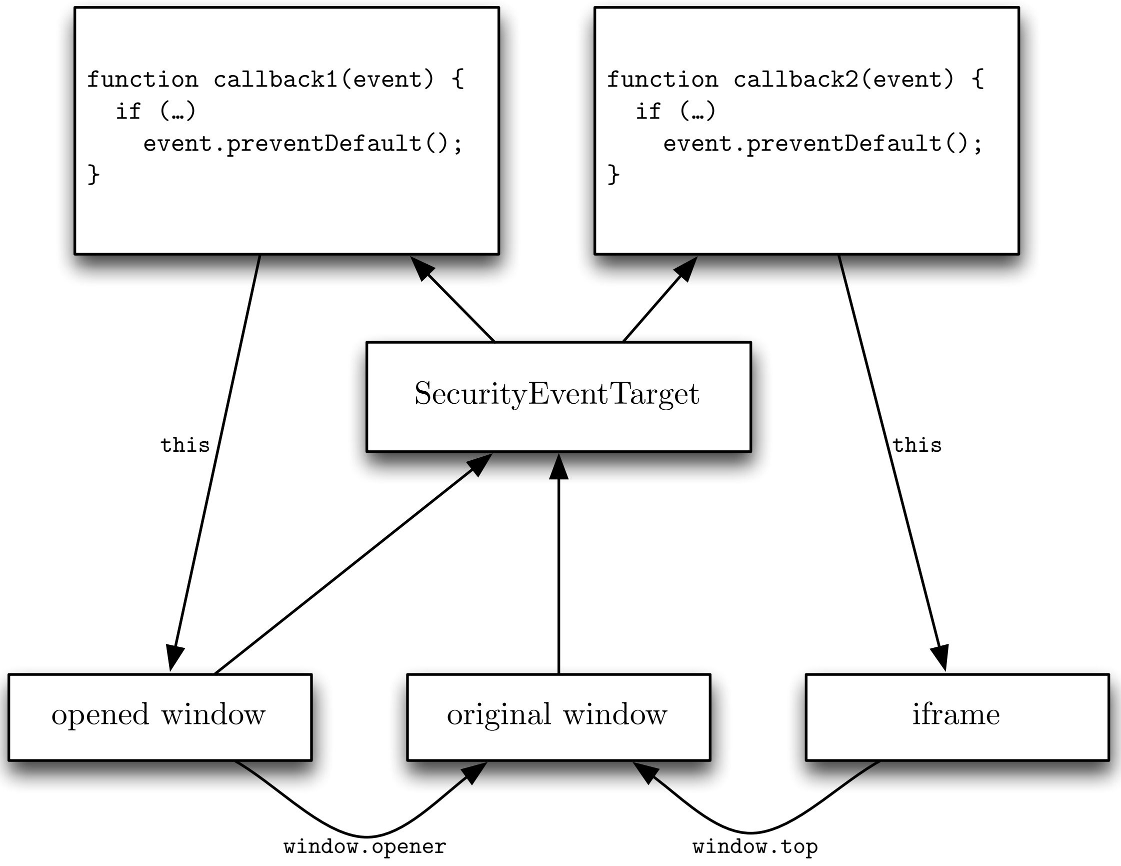
6.2.4 Default labels

All of this tagging infrastructure works quite well for data that can be tagged when it is loaded, but it does not help with data that is both sensitive and built-in to the browser.

Like Netscape, then, I've added a couple of default tags to the browser. `document.urlTag` is applied to `document.URL` and to the fields of the `Location` object. `document.cookieTag` is similarly applied to the `cookie` field. Those tags are in immutable fields but globally accessible, so while there won't be false negatives, any code could add those tags to data.

In order to allow some additional decision-making ability, every document also comes with its own default tag, accessible via `document.securityTag`. Every node upon creation must belong to a document, so it copies the document's security label to itself (which includes that tag). Again, the default tag won't have false negatives, but it could be added from anywhere that the `document` object can be accessed.

Figure 6.3 One `SecurityEventTarget` in use by three frames



Here we have one `SecurityEventTarget` in use by three frames. It originally belonged to the original window. When that window created an `iframe` that added a security event handler, it added it to its parent's target. Finally, when the original window opened another window via JavaScript, it set the new window's target to its own.

Chapter 7

Evaluation

I will evaluate my work under three categories: completeness, performance, and usefulness.

7.1 Completeness

Because WebKit is a massive project and because I am just one person, I make no guarantees for the completeness of my reference monitor. Certainly, I am aware of areas where data can be laundered to strip its label, and it is likely that there are areas I think are safe that can be manipulated to make them unsafe. For example, my labeling modifications to JavaScriptCore do not track control flow or indirect control flow. Both of these were solved by Vogt et al., but require a good amount of additional work, including linear static analysis of bytecode.

Were this kind of a labeling system to be shipped in a browser, however, a large team would be able to go through and verify complete tracking and mediation up to a point. The complex interactions between features can lead to security holes that are quite difficult to find. This is true even in today's browsers, where holes in the same-origin policy have been found even after a decade and a half of its existence [9].

Nevertheless, I do have a test suite taken from TaintSNIFFER [21] that tests to make sure that all sorts of JavaScript operations propagate labels, ranging from addition to object property access to calling functions. The test itself is run 100000 times in order to hit the different optimization paths of the JavaScript engine, since each engine's code generation is activated when a code path gets sufficiently hot as to warrant spending time in a code generator.

These tests do not cover all operations, primarily because some are simply not implemented. The JavaScript evaluation function `eval`, for example, will strip off labels. A solution to this is to add a current label to the interpreter that should be merged into all newly created objects and assignments—the same solution that is needed for propagating labels through control flow dependencies. This is quite doable, but it requires yet more work to be done.

I also have implemented a series of WebKit layout tests that test for both label propagation between the DOM and JavaScript and that the events that I created behave properly. “Layout test” is a bit of a misnomer, since the term includes tests that test everything in WebKit, not just layout. By implementing those tests, I both provided myself assurance that the features work and made a further step towards where the WebKit team could merge my work in if they chose to do so.

7.2 Performance

One major goal in any tainting, tagging, or labeling implementation is to have reasonable performance. These systems are designed to solve real-world problems that do not have good preexisting solutions, but if the cost is unresponsive code, then no one will use them.

Because my implementation is done in the application’s original source, not by dynamically rewriting binaries or by using hardware simulators, I expected the overhead of my work to be reasonable enough that, with further work, it could be merged into WebKit and shipped. Indeed, that is the case.

In terms of user-perceived performance, the overhead is not bad. As shown in Appendix A, comparing runs of the different performance tests included in WebKit show that the tests on my branch is at most 1.7 times as slow as on the unchanged master branch. This is almost entirely because of the default labels that I added to every node, so with empty labels performance is similar.

Those tests almost all measure interaction between JavaScript and WebCore, so it does not solely represent the impact of adding label propagation to the JavaScript engine. A decent approximation of the overhead in the engine itself is the time difference between running with no labels added on both the master branch and my branch. As shown in Table 7.1, the overhead is around 1.1x in the engine itself, and another 1.5x for labeling. A similar comparison using the SunSpider suite gives similar results, as shown in Table 7.2.

Branch	Time
Master branch (null labeling)	1.352s
My changes (null labeling)	1.504s
My changes (labeling enabled)	2.275s

Table 7.1: Performance in TaintSNIFFER suite

Branch	Time
Master branch	140.9ms
My changes	161.0ms

Table 7.2: SunSpider performance

With further work, I am confident that the overhead can be brought down significantly. The lowest-hanging fruit to reduce it would be to modify the DFG JIT, the JIT, and the low-level interpreter in JavaScriptCore to handle labels directly in assembly. Currently, when they see a labeled value, they call into `C++` code. With those changes, more speculative optimization would be possible as well. I was forced to change the predicted results of operations to account for labeled values, which in turn reduced the effectiveness of the speculative DFG JIT.

7.3 Usefulness

In order to demonstrate my framework's capabilities, I built a small web application that loads an external script that attempts to steal data. It attempts to steal that data in the following ways:

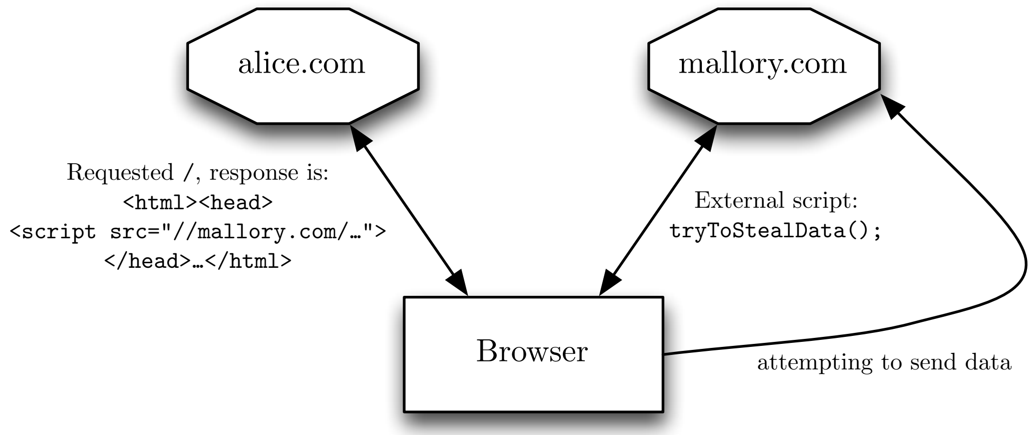
- loading an image with the data in the URL
- loading a script with the data in the URL
- loading a stylesheet with the data in the URL
- loading an iframe that then posts the sensitive data off

Its architecture is illustrated in Figure 7.1. The script loaded from `mallory.com` tries to send that data back home, and succeeds in every case using the stock WebKit. With my changes, though, the JavaScript in Figure 7.2 is sufficient to protect the sensitive data.

All that is required is that this JavaScript code executes before any other JavaScript does, which is trivial. Minified, the code is less than 500 bytes, even with a few guards and a couple more events added in, so it could even be included inline. With a little more work, a wrapper function could be added to allow transmission of that sensitive data from certain specific functions using a boolean flag in both the closure of the monitor function and the sending function.

The reference monitor can clearly be used to prevent data leaks, but it has other uses as well. For example, suppose that a developer knew that all AJAX requests would originate in a fixed number of places in the code base. Then, it would be quite simple for her to add a monitor function that would prevent all requests when a flag like the one above is

Figure 7.1 The basic architecture of the test application



disabled. This kind of work doesn't even require labeling, just the addition of useful hooks to the browser. The addition of labeling just expands the space of useful policies that can be enforced.

One addition that would expand the space of policies that could be enforced would be to add an integrity label to objects, as well as the label that I've added already. Flume [10] proposes integrity labels as distinct from secrecy labels, although it does not actually implement them. I envision them as labels that do not survive operations, acting as assurance that object have not changed since a tag was applied.

Figure 7.2 Preventing Mallory from stealing data

```
(function () {
  var tag = new SecurityTag();
  var events = ['checkxhropen', 'checkxhrsend',
    'checkbeforeload', 'checknavigate'];
  var eventName;
  for (var index in events) {
    eventName = events[index];
    window.addEventListener(eventName, function (event) {
      if (tag.isOn(event.securityLabel))
        event.preventDefault();
    });
  }

  // make a read-only undeletable reference to the tag
  Object.defineProperty(window, 'privateTag', {
    value: tag
  });
})();
```

This snippet of code adds security event handlers to the events listed in `events`, which simply block the default if the tag is present on the label passed into the event handler. It then defines a unmodifiable property on the global object that points to the tag object, ensuring that all code that needs to mark sensitive data is assured that `privateTag` has handlers defined to check for it. Note that this code is executed in an anonymous function, which defines a new scope that the event handlers can close over.

Chapter 8

Conclusion

As a result of its somewhat ad-hoc development over the past two decades, the browser still does not have more than the basics of a security policy, and the one that it does have is overly strict. Instead of coming up with a new security model that would be both more powerful and less restrictive of legitimate activities, browser vendors have chosen the path of adding more complexity to allow for cross-domain access [14].

It's easy to understand why they chose that path—a wholesale change could backfire by putting existing sites at risk. That's also why I kept enforcing the same-origin policy with my changes, but it was not without regret. What I would like to see is a way to let sites opt in to only use the labeling reference monitor model.

Even without that ability, the layering of the two security models is certainly better than just the same-origin policy alone. As I've shown in Chapter 7, it's quite easy to steal data if a site loads an external script. When it unintentionally loads that script, that's a cross-site scripting attack, so this reference monitor can defend against that.

More valuable than just defending against a certain class of attacks, though, is the power of hooks. Hooks in the right places can allow future developers to defend against unforeseen attacks, as well as to do all sorts of things that I can't predict. The existing hooks in the browser have allowed all sorts of web development to occur that browser developers probably did not predict, so I hope that the addition of security hooks and data flow tracking will allow further improvements in browser security.

Bibliography

[1] Georges Auberger. *Secure Cross Domain iFrame Communication.* Ternary Labs. Mar. 27, 2011. URL: `http://ternarylabs.com/2011/03/27/secure-cross-domain-iframe-communication/`.

[2] Adam Barth. *HTTP State Management Mechanism. RFC 6265.* Internet Engineering Task Force. Apr. 2011. URL: `http://tools.ietf.org/rfc/rfc6265.txt`.

[3] Netscape Communications Corporation. *JavaScript Guide.* URL: `http://web.archive.org/web/20060318153542/wp.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html`.

[4] Netscape Communications Corporation. *What's New in JavaScript 1.2.* 1997. URL: `http://web.archive.org/web/19971015223714/http://developer.netscape.com/library/documentation/communicator/jsguide/js1_2.htm`.

[5] Julian Couvreur. *Cross-document Messaging Hack.* Sept. 18, 2006. URL: `http://blog.monstuff.com/archives/000304.html`.

[6] Michael Dalton, Hari Kannan, and Christos Kozyrakis. "Raksha: A Flexible Information Flow Architecture for Software Security". In: *In International Symposium on Computer Architecture (ISCA.* 2007.

[7] Brendan Eich. *Popularity.* 2008. URL: `http://brendaneich.com/2008/04/popularity/`.

[8] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1. RFC 2616.* Internet Engineering Task Force. Apr. 2011. URL: `http://tools.ietf.org/rfc/rfc6265.txt`.

[9] Serg Glazunov. *Cross-origin Access Using `window.execScript` + Code Execution.* May 18, 2011. URL: `http://code.google.com/p/chromium/issues/detail?id=83096`.

[10] William R. Harris et al. "Verifying Information Flow Control over Unbounded Processes". In: *Proceedings of the 2nd World Congress on Formal Methods.* FM '09. Eindhoven, The Netherlands: Springer-Verlag, 2009, pp. 773-789. ISBN: 978-3-642-05088-6. DOI: `10.1007/978-3-642-05089-3_49`. URL: `http://dx.doi.org/10.1007/978-3-642-05089-3_49`.

[11] Ian Hickson, ed. *HTML5*. World Wide Web Consortium. 2012. URL: `http://www.w3.org/TR/html5/`.

[12] ECMA International. *ECMAScript Language Specification*. Version 5.1 Edition. June 2011.

[13] Vasileios P. Kemerlis et al. "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. VEE '12. New York, NY, USA: ACM, 2012, pp. 121-132. ISBN: 978-1-4503-1176-2. DOI: `10.1145/2151024.2151042`. URL: `http://doi.acm.org/10.1145/2151024.2151042`.

[14] Anne van Kesteren, ed. *Cross-Origin Resource Sharing*. World Wide Web Consortium. Apr. 3, 2012. URL: `http://www.w3.org/TR/cors/`.

[15] Anne van Kesteren, ed. *XMLHttpRequest Level 2*. World Wide Web Consortium. Jan. 12, 2012. URL: `http://www.w3.org/TR/XMLHttpRequest/`.

[16] Anne van Kesteren, Aryeh Gregor, and Ms2ger, eds. *DOM4*. World Wide Web Consortium. 2012. URL: `http://www.w3.org/TR/2012/WD-dom-20120405/`.

[17] Butler W. Lampson. "Protection". In: *SIGOPS Oper. Syst. Rev.* 8.1 (Jan. 1974), pp. 18-24. ISSN: 0163-5980. DOI: `10.1145/775265.775268`. URL: `http://doi.acm.org/10.1145/775265.775268`.

[18] Benjamin S. Lerner et al. "C3: An Experimental, Extensible, Reconfigurable Platform for HTML-based Applications". In: *2nd USENIX Conference on Web Application Development*. University of Washington & Microsoft Research. USENIX, 2011. URL: `http://research.microsoft.com/apps/pubs/default.aspx?id=150010`.

[19] Cameron McCormack, ed. *Web IDL. W3C Candidate Recommendation*. World Wide Web Consortium. Apr. 19, 2012. URL: `http://www.w3.org/TR/WebIDL/`.

[20] Don Melton. *Greetings from the Safari team at Apple Computer*. Forwarded to the `kfm-devel` mailing list by Dirk Mueller. Jan. 7, 2003. URL: `http://lists.kde.org/?l=kfm-devel&m=104197092318639&w=2`.

[21] Aaron Miller and Paramijt Singh Sandhu. "TaintSNIFFER: A Robust Dynamic Taint Tracking System For a Homogenous Web Browsing Environment". University of Washington CSE 501 final project. 2010. URL: `http://www.cs.washington.edu/education/courses/cse501/10au/TaintSNIFFER.pdf`.

[22] John Resig. *jQuery*. 2012. URL: `https://github.com/jquery/jquery/blob/247d824d35e81ec96e1a8913674c97fb45dd40ae/src/core.js#L368`.

[23] Stephen Smalley, Chris Vance, and Wayne Salamon. *Implementing SELinux as a Linux Security Module*. Tech. rep. National Security Agency, 2006.

[24] Shuo Tang et al. "Alhambra: A System for Creating, Enforcing, and Testing Browser Security Policies". In: *Proceedings of the 19th international conference on World wide web*. WWW '10. New York, NY, USA: ACM, 2010, pp. 941-950. ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772786. URL: http://doi.acm.org/10.1145/1772690.1772786.

[25] United States Department of Defense. *Department of Defense Trusted Computer Evaluation Criteria*. (Orange Book). Dec. 1985.

[26] Philipp Vogt et al. "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In: *NDSS*. San Diego, 2007.

[27] WebKit Open Source Project. *WebKit*. 2012. URL: http://www.webkit.org/.

[28] Andy Wingo. *JavaScriptCore, the WebKit JS implementation*. Oct. 30, 2011. URL: http://wingolog.org/archives/2011/10/28/javascriptcore-the-webkit-js-implementation.

[29] Alexander Yip et al. "Privacy-preserving Browser-side Scripting with BFlow". In: *Proceedings of the 4th ACM European conference on Computer systems*. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 233-246. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519091. URL: http://doi.acm.org/10.1145/1519065.1519091.

[30] Michal Zalewski. *Browser Security Handbook*. 2010. URL: http://code.google.com/p/browsersec/wiki/Main.

[31] David (Yu) Zhu et al. "TaintEraser: protecting sensitive data leaks using application-level taint tracking". In: *SIGOPS Oper. Syst. Rev.* 45.1 (Feb. 2011), pp. 142-154. ISSN: 0163-5980. DOI: 10.1145/1945023.1945039. URL: http://doi.acm.org/10.1145/1945023.1945039.

Appendix A

Performance test results

Here are the results from comparing runs of WebKit's performance tests:

Test	Overhead	Master branch	My changes
Bindings			
dom-attributes	1.143x	2676.0 ms ±0.7%	3058.4 ms ±0.4%
event-target-wrapper	1.011x	550.3 ms ±0.4%	556.4 ms ±0.2%
CSS			
CSSPropertySetterGetter	1.178x	953.8 ms ±0.2%	1123.5 ms ±0.2%
CSSPropertyUpdateValue	1.145x	846.1 ms ±0.2%	969.1 ms ±0.1%
DOM			
Accessors	1.32x	170.1 ms ±0.3%	224.4 ms ±1.8%
CloneNodes	1.084x	183.9 ms ±0.4%	199.2 ms ±0.8%
CreateNodes	1.109x	233.0 ms ±0.3%	258.5 ms ±1.3%
DOMDivWalk	1.118x	53.7 ms ±1.1%	60.0 ms ±1.3%
DOMTable	-	846.1 ms ±1.4%	840.8 ms ±0.8%
DOMWalk	1.143x	0.2 ms ±3.2%	0.2 ms ±2.6%
Events	1.036x	202.0 ms ±2.2%	209.3 ms ±2.3%
GetElement	0.979x	496.8 ms ±0.7%	486.3 ms ±0.5%
GridSort	1.109x	16.1 ms ±1.3%	17.8 ms ±1.4%
Template	1.138x	7.9 ms ±1.7%	9.0 ms ±1.0%
Dromaeo			
cssquery-dojo	1.090x	277348.9 runs/s ±0.0%	254422.8 runs/s ±0.0%
cssquery-jquery	1.24x	4213097.2 runs/s ±0.1%	3390151.6 runs/s ±0.1%
cssquery-prototype	1.060x	338425.4 runs/s ±0.0%	319316.0 runs/s ±0.0%
dom-attr	1.34x	5631.2 runs/s ±0.9%	4196.8 runs/s ±0.1%
dom-modify	1.057x	5470.3 runs/s ±0.4%	5177.1 runs/s ±0.3%
dom-query	1.20x	812493.5 runs/s ±0.2%	675873.6 runs/s ±0.5%
dom-traverse	1.047x	3566.1 runs/s ±0.1%	3405.2 runs/s ±0.1%
dromaeo-3d-cube	1.015x	1841.0 runs/s ±1.2%	1814.0 runs/s ±0.9%
dromaeo-core-eval	1.30x	1514.7 runs/s ±0.4%	1166.0 runs/s ±0.3%
dromaeo-object-array	1.003x	48774.0 runs/s ±0.1%	48606.3 runs/s ±0.1%
dromaeo-object-regexp	1.023x	10600.9 runs/s ±0.0%	10362.4 runs/s ±0.0%
dromaeo-object-string	1.40x	80745.2 runs/s ±0.1%	57494.5 runs/s ±0.2%
dromaeo-string-base64	1.48x	2016.2 runs/s ±0.9%	1361.9 runs/s ±1.7%
jslib-attr-jquery	1.56x	9024.1 runs/s ±0.5%	5793.9 runs/s ±0.3%
jslib-attr-prototype	1.34x	6582.7 runs/s ±0.2%	4917.5 runs/s ±0.5%
jslib-event-jquery	1.195x	1102.0 runs/s ±12.5%	922.0 runs/s ±6.2%

Test	Overhead	Master branch	My changes
jslib-event-prototype	1.184x	2070.5 runs/s ±0.3%	1749.4 runs/s ±0.5%
jslib-modify-jquery	1.42x	12976.5 runs/s ±2.8%	9153.9 runs/s ±4.1%
jslib-modify-prototype	1.071x	3251.6 runs/s ±0.3%	3037.4 runs/s ±0.4%
jslib-style-jquery	1.25x	5762.5 runs/s ±0.4%	4620.6 runs/s ±0.2%
jslib-style-prototype	1.145x	7725.8 runs/s ±0.1%	6746.4 runs/s ±0.2%
jslib-traverse-jquery	1.083x	3050.7 runs/s ±0.1%	2815.9 runs/s ±0.2%
jslib-traverse-prototype	1.167x	9353.7 runs/s ±0.2%	8012.3 runs/s ±0.1%
sunspider-crypto-md5	-	2413.6 runs/s ±9.8%	2384.8 runs/s ±3.5%
sunspider-crypto-sha1	0.910x	2403.0 runs/s ±5.6%	2641.8 runs/s ±1.7%
Layout			
floats_100_100	1.054x	138.0 ms ±6.5%	145.5 ms ±9.3%
floats_100_100_nested	1.023x (inconclusive)	150.0 ms ±6.0%	153.5 ms ±8.8%
floats_20_100	1.005x (inconclusive)	308.0 ms ±1.3%	309.4 ms ±1.9%
floats_20_100_nested	1.013x (inconclusive)	461.8 ms ±1.2%	467.7 ms ±3.5%
floats_2_100	1.022x	140.5 ms ±0.9%	143.6 ms ±1.3%
floats_2_100_nested	0.973x	372.4 ms ±0.8%	362.2 ms ±1.4%
floats_50_100	-	209.0 ms ±1.3%	208.8 ms ±0.8%
floats_50_100_nested	1.009x (inconclusive)	255.6 ms ±0.8%	257.8 ms ±2.2%
PageLoad			
42450-under the see.svg	1.052x (inconclusive)	131.2 ms ±6.9%	138.1 ms ±8.2%
42470-flower_from_my_garden_v2.svg	1.059x (inconclusive)	66.3 ms ±4.7%	70.2 ms ±6.4%
44057-drops on a blade.svg	1.045x (inconclusive)	113.8 ms ±33.0%	118.9 ms ±32.6%
Harvey_Rayner.svg	1.101x (inconclusive)	86.9 ms ±8.2%	95.8 ms ±6.0%
bamboo_01.svg	1.054x (inconclusive)	1065.6 ms ±3.0%	1123.4 ms ±7.7%
cacuts_01.svg	1.064x (inconclusive)	163.4 ms ±4.7%	173.9 ms ±7.5%
cowboy.svg	1.043x (inconclusive)	263.4 ms ±3.0%	274.8 ms ±4.9%
crawfish2_ganson.svg	1.036x (inconclusive)	69.9 ms ±7.5%	72.3 ms ±10.2%
deb9frac1.svg	1.032x (inconclusive)	139.8 ms ±10.4%	144.2 ms ±9.4%
food_leif_lodahl_01.svg	1.114x	109.5 ms ±6.9%	122.0 ms ±7.4%
world-iso.svg	1.057x	540.1 ms ±1.0%	570.7 ms ±2.1%
worldcup.svg	1.006x (inconclusive)	2816.1 ms ±3.0%	2834.4 ms ±3.5%
Parser			
css-parser-yui	1.045x	241.7 ms ±0.2%	252.5 ms ±0.5%
html-parser	1.006x	1366.7 ms ±0.1%	1375.3 ms ±0.8%
html5-full-render	1.020x	16145.0 ms ±1.4%	16475.5 ms ±1.7%
simple-url	1.38x	179.5 ms ±1.4%	247.3 ms ±0.9%
tiny-innerHTML	1.115x	1420.0 ms ±0.3%	1584.0 ms ±0.3%
url-parser	1.40x	78.9 ms ±0.4%	110.3 ms ±0.5%
xml-parser	1.086x	1258.2 ms ±0.2%	1366.5 ms ±0.2%

Appendix B

Data structures

B.1 `SecurityTag`

Figure B.1 `SecurityTag`

```
struct SecurityTag {
public:
    SecurityTag() : m_impl(monotonicallyIncreasingTime()) {}
    operator double() const { return m_impl; }
private:
    double m_impl;
};
```

`SecurityTags` is basically just a protected `typedef` wrapping a `double`.

B.2 `SecurityTagObject`

A `SecurityTagObject` is the object seen in JavaScript as a `SecurityTag`. In JavaScript-Core, an object's structure is initialized when the `JSGlobalData` object for that thread is created. At that time, the structure for the `SecurityTagObject`, the `SecurityTagProto-type`, and the `SecurityTagConstructor` are frozen. That ensures that no properties can be added, removed, or changed on any `SecurityTag` instance in JavaScript. At the same time, the definition of the `SecurityTag` property on the global object is frozen, preventing it from being changed or erased. Consequently, there is no way for any attack to tamper with the tagging infrastructure.

Its methods, `addTo` and `isOn`, ask each `JSCell` (the base class for every non-immediate JavaScript value) for the appropriate data through a wrapper method that calls a method from a class-specific static table. Since JSCells do not use virtual methods, this is the only way to allow different JavaScript objects to handle labeling differently.

B.3 SecurityLabel

Figure B.2 SecurityLabel

```
class SecurityLabel {
    //...
private:
    RefPtr<SecurityLabelImpl> m_impl;
};
```

Figure B.3 SecurityLabelImpl

```
  class SecurityLabelImpl : public RefCounted<SecurityLabelImpl> {
  private:
      typedef HashSet<SecurityTag> SecurityTagSet;
  public:
      //...
  private:
      RefPtr<StringImpl> m_descriptor;
      SecurityTagSet m_tagSet;
      HashMap<RefPtr<StringImpl>, RefPtr<StringImpl> > m_transitionTable;
      HashMap<SecurityTag, RefPtr<StringImpl> > m_tagTransitionTable;
      bool m_setCreated;
  };
```

A `SecurityLabel` (Figure B.2) is a class that only contains a reference-counted pointer to a `SecurityLabelImpl` (Figure B.3), the class that actually implements a individual label.

A label is a set of `SecurityTag`s, which are just `double`s initialized with the current time. Initially, I simply stored labels as `HashSet`s of tags, but this made every label merge operation allocated and produce a new label. Ideally, the merge operation on two sets a and b would produce a pointer to the same label object every time.

Therefore, I added a string field `descriptor` that describes every label. It's simply a concatenation of the sorted tags in their 64-bit `double` representation. WebKit already has an infrastructure for caching and hashing string implementations, so it's trivial to keep a mapping from description to label implementation object. Finally, each label caches the transitions that it encounters by caching the descriptor of the resulting label. That allows the resulting label to be reconstructed if it was destroyed in the meantime, in addition to avoid circular references that would cause a memory leak.

B.4 SecurityLabelObject

Keeping track of labels on non-JavaScript objects is quite easy, then, because it just requires adding a `SecurityLabel` field. The JavaScript wrapper then just needs to point to that label. Pure JavaScript objects, however, have a garbage-collection related problem. All JavaScript objects and strings are `JSCell`s, which are allocated and collected by JavaScriptCore's pre-allocated heap.

Figure B.4 `JSCell` [27]

```
class JSCell {
    //...
    private:
      const ClassInfo* m_classInfo;
        WriteBarrier<SecurityLabelObject> m_label;
        WriteBarrier<Structure> m_structure;
// 32-bit systems need to force other fields to be double-aligned
#if USE(JSVALUE32_64)
        void* m_blank;
#endif
}
```

Most of the subclasses of `JSCell` do not require that a destructor be called on them when they are destroyed, as all of their references are to other garbage-collected objects. There are even assertions to that effect in their code. Adding a reference-counted field to `JSCell` would force every cell to be destroyed with a destructor.

Instead of forcing that change, I added a new primitive JavaScript type, `SecurityLabelObject` that wraps the reference-counting needed by `SecurityLabel`. Every `JSCell` (Figure B.4) has a `WriteBarrier`-based reference to that wrapper type. That `WriteBarrier` lets the garbage-collector know that each cell may have a reference to a label object while also not needing a destructor. Similar to the table mapping label descriptors to label objects, I added a table mapping `SecurityLabelImpl*` to `SecurityLabelObject`. That ensures that there is only one JavaScript wrapper for each label at any given time.

Conveniently enough, this also is the type used when dispatching `SecurityEvents`. Since it is a primitive, no properties can be added to it. I also redefined its labeling method (`mergeSecurityLabelCell`) to ignore any merge requests, so that it can't be modified at all.

B.5 JSLabeledValue

As I discussed in Section 5.4, there isn't space for labels in the representation of immediate primitive values. One possible solution to this, of course, is to enlarge the size

Figure B.5 `EncodedJSValue` [27]

```
#if USE(JSVALUE32_64) // 32-bit systems
    typedef int64_t EncodedJSValue;
#else
    typedef void* EncodedJSValue;
#endif
```

of `EncodedJSValue` (whose current representation is shown in Figure B.5). That choice, however, would force virtually every line of assembly-generating code to be changed.

Instead, I created another primitive type, `JSLabeledValue`. It is a `JSCell` that has an additional field that stores a `JSValue`. Because it holds only immediates, it doesn't even need to let the garbage collector know that it contains anything that a base `JSCell` doesn't.

Appendix C

`SecurityEvent`

Figure C.1 The `SecurityEvent` interface

```
interface SecurityEvent : Event {
    readonly attribute DOMString destination;
    readonly attribute DOMWindow source;
    readonly attribute DOMObject securityLabel;
}
```

A `SecurityEvent` has four useful fields, as shown in Figure C.1. The destination is the URL to which data will be sent, if there is such a URL, while the window is the context in which the request is being made if it exists. Both of those may be null if they don't make sense for the event or if the event handler receiving the event would not normally be able to inspect the requester's frame.

The generation of the `SecurityLabel` is a little bit more interesting. One consequence of how I have implemented labeling is that a parent object that contains a labeled object will not necessarily have the child's label on it. Figuring out how to walk various different kinds of objects to determine their labeling would be messy, so I instead generate the label from the object's serialization.

That serialization is what is going to be leave the system, so all I needed to do was ensure that the label of a serialization of an object is the accumulation of everything that went into it—a guarantee that is somewhat easy to make given that much of that serialization would happen in JavaScript.