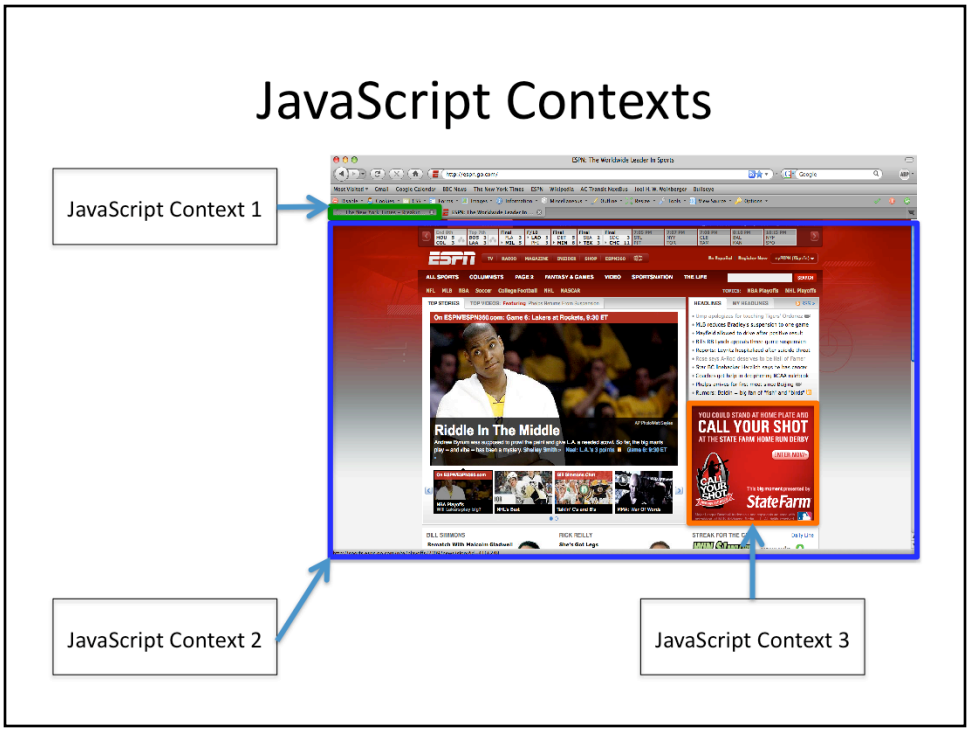
Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense

Adam Barth
Joel Weinberger
Dawn Song
University of California, Berkeley

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

Joint work with Adam Barth and Dawn Song



JavaScript is a simple language with complex security properties. Specifically, it is concerned about hostile code being run in a variety of JavaScript contexts. Take this example. We have (at least) three distinct JavaScript contexts: the ESPN page, an advertisement running in a frame, and NYTimes.com running in another tab. All of these could be running JavaScript.

JavaScript objects from one JavaScript context should not necessarily be accessible from another JavaScript context. This could lead to all sorts of malicious behavior such as accessing another site's cookies or changing the JavaScript of that page. In this work, we're particularly worried about a class of vulnerabilities that leaks JavaScript objects from one JavaScript context to another.

In particular, are there ways for one context to maliciously access objects and properties in another context?

Contributions

- Identify new class of browser vulnerabilities

In this work, we identify a new class of web browser security vulnerabilities which allow for the access of objects and properties in other JavaScript contexts. These vulnerabilities exploit a particular hole in the security *enforcement* by web browsers of their security policies. We call these vulnerabilities “Cross-Origin JavaScript Capability Leaks.”

Contributions

- Identify new class of browser vulnerabilities
- A dynamic tool for detecting these bugs

We also have created a dynamic analysis tool for detecting these vulnerabilities. We use a novel form of JavaScript heap graph analysis to accomplish this.

Contributions

- Identify new class of browser vulnerabilities
- A dynamic tool for detecting these bugs
- Discover several real vulnerabilities

Using the tool, we find two several real vulnerabilities in a major web browser. Additionally, we also use to the tool to dissect a “safe” mashup JavaScript library and exploit it.

Contributions

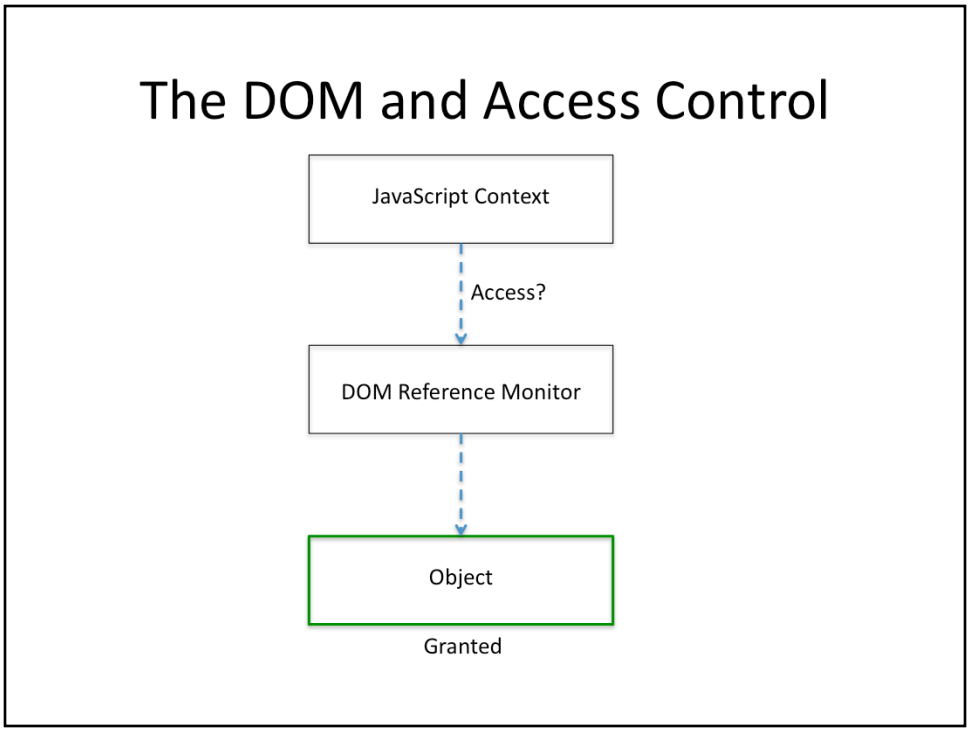
- Identify new class of browser vulnerabilities
- A dynamic tool for detecting these bugs
- Discover several real vulnerabilities
- A new enforcement mechanism for browser security policies

Finally, we propose a new enforcement mechanism for web browsers. We do *not* propose a new policy; we only propose a new, more effective, enforcement mechanism for current policies.

Overview

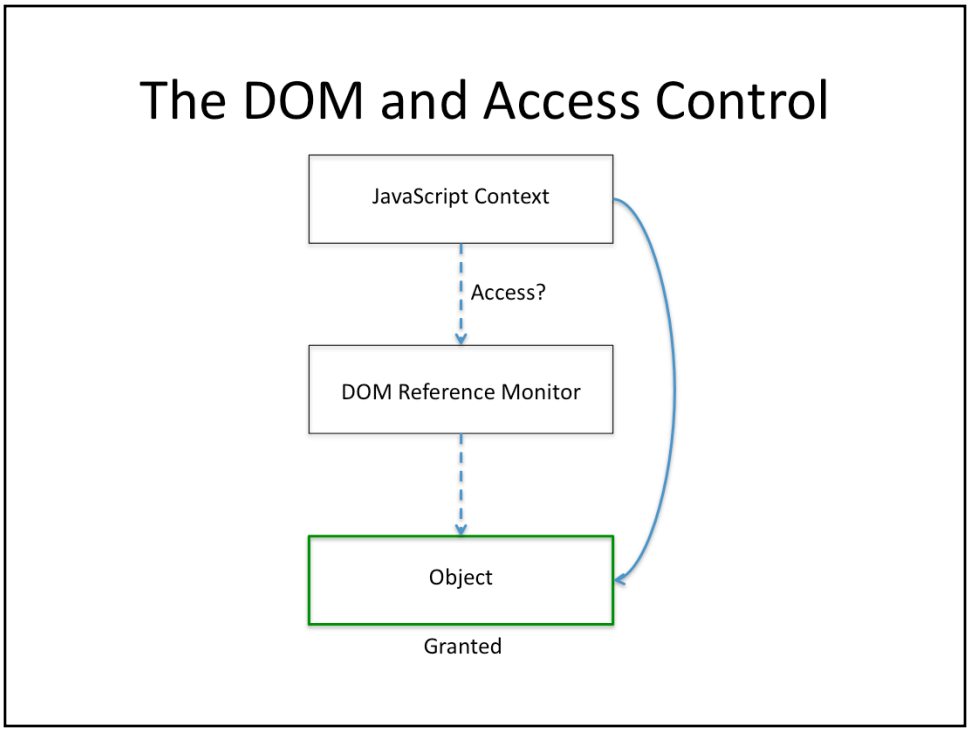
- Current JavaScript Security Model
- Cross-Origin JavaScript Capability Leaks
- Capability Leak Detection
- Browser Defense Mechanism

To start the talk, let's discuss the current JavaScript security model for object access. Then, we'll introduce the problem of Cross-Origin JavaScript Capability Leaks. We'll show a method of detecting these vulnerabilities. Finally, we'll discuss a general solution to this class of attacks.

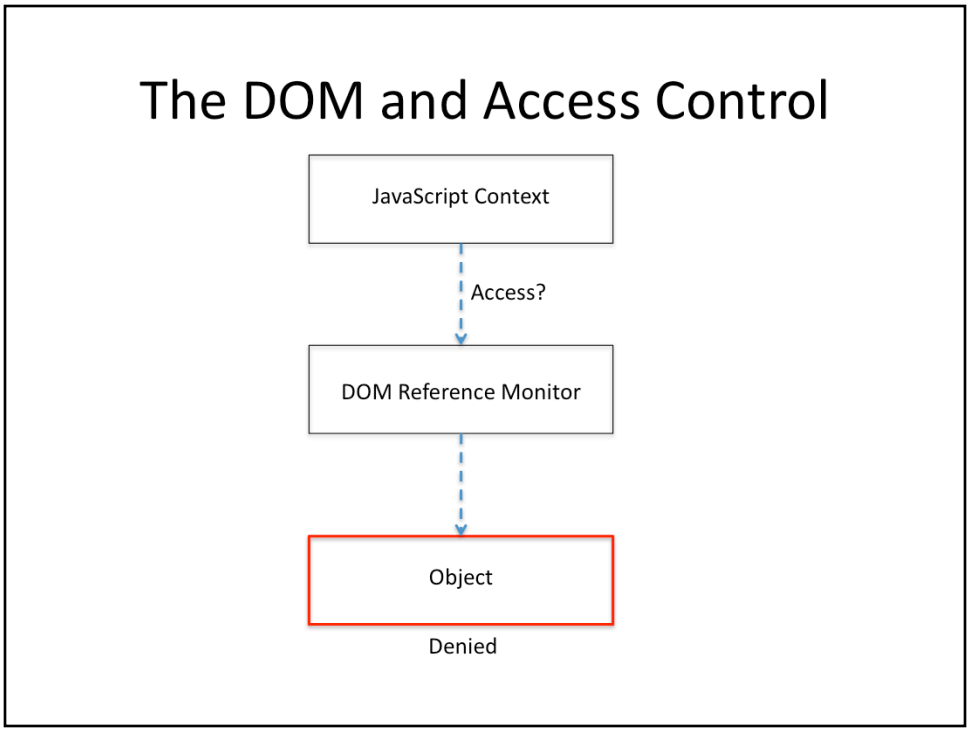


The Document Object Model, or DOM, is the structure that represents many of the important objects on web pages, such as the document's cookie. It also allows for the physical manipulation of the web page itself. The DOM is not directly a part of the JavaScript engine; it is a set of built in objects and methods for manipulating objects, but the JavaScript engine is theoretically separate from the DOM.

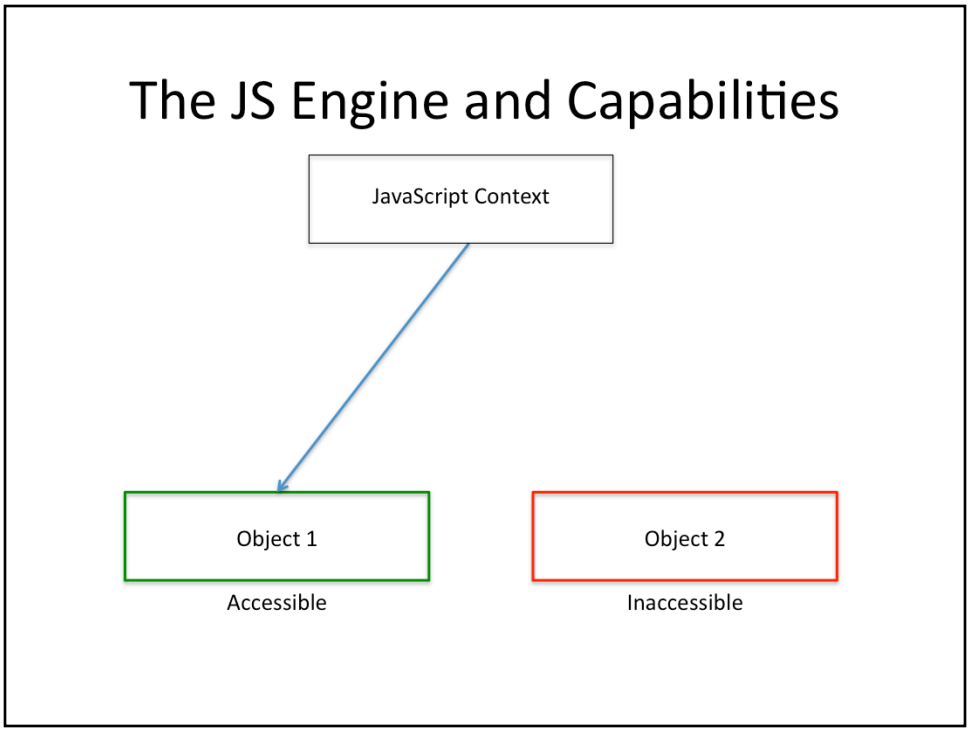
In order to gain access to DOM objects, the DOM does a security check to make sure that the accessing context is allowed to handle the specified object. If the JavaScript contexts match, the connection is granted and access given.



From the JS Engine's perspective, the JavaScript context now holds a reference to the object.



If the JavaScript contexts do not match, then access is denied and no reference is given.



The JavaScript engine itself has a different way of doing things. It works as a capability system. If a JavaScript context is given a reference to a JS object, it has permission to access it. If no such reference exists, the object cannot be accessed. There is no way to “divine” objects in the JavaScript engine. This is sort of where the DOM comes in. If you need access to a DOM object, you can reference it, even if no particular object has a reference to it.

DOM vs JS Engine

- The DOM provides an access control layer

In short, inside of web browsers, there are two different ways mechanisms for security. On the one hand, the DOM provides access control checks when a DOM object is initially accessed.

DOM vs JS Engine

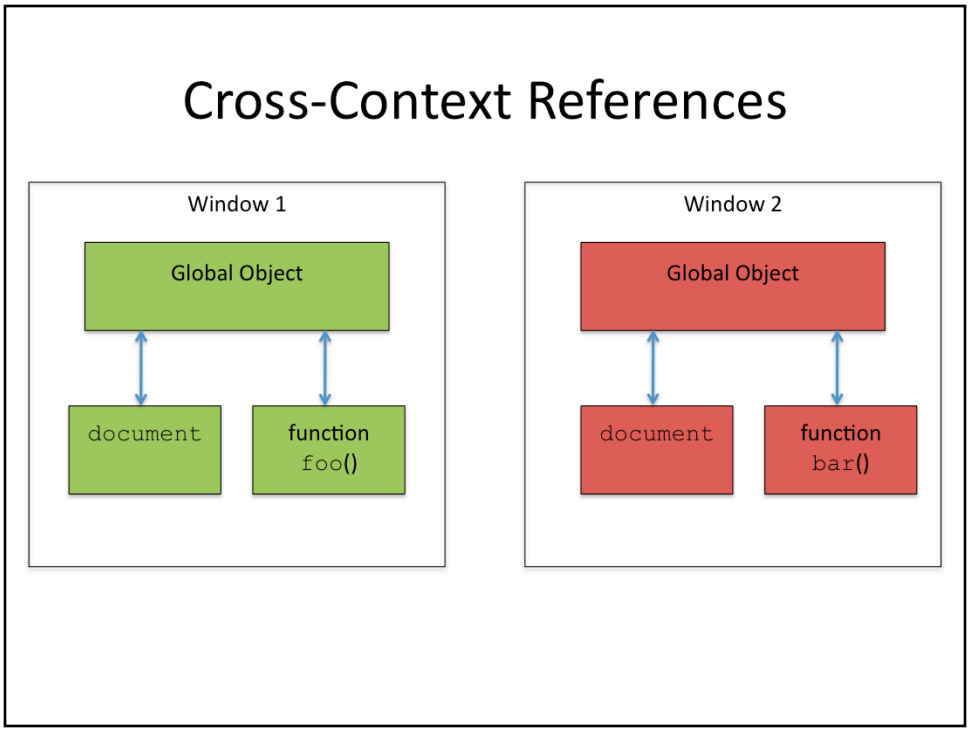
- The DOM provides an access control layer
- The JavaScript engine treats objects as capabilities

On the other hand, the JavaScript engine treats all objects as capabilities, including DOM objects once they have been accessed and assigned a variable.

Overview

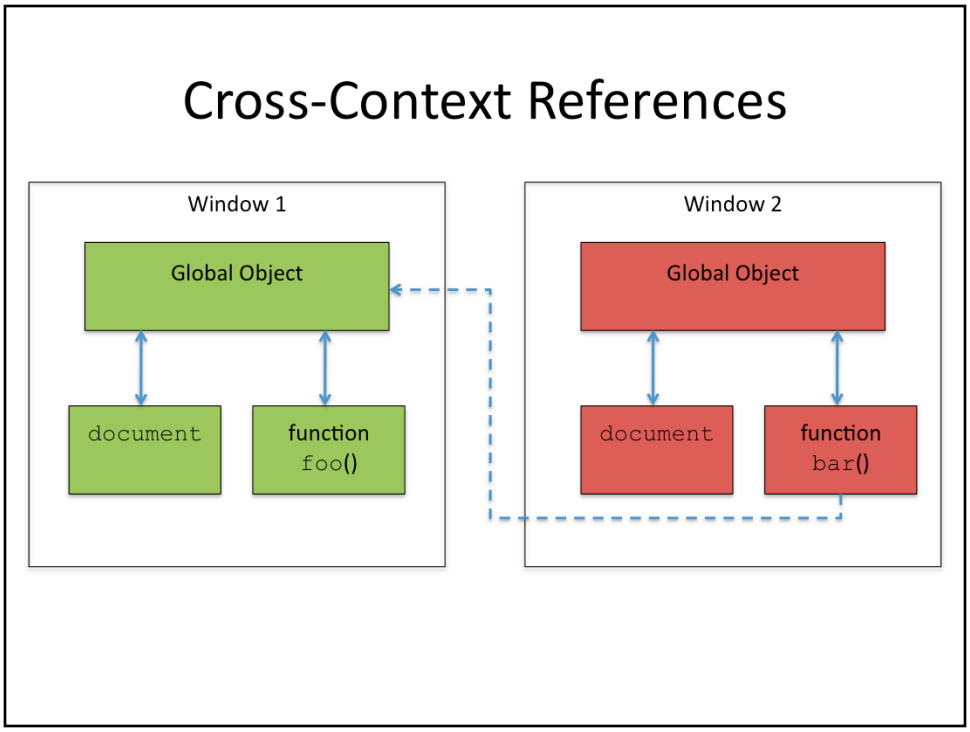
- Current JavaScript Security Model
- Cross-Origin JavaScript Capability Leaks
- Capability Leak Detection
- Browser Defense Mechanism

You might start to get a sense that this situation is a bit odd. We have the DOM acting as an access control system and the JS Engine as a capability system, both of which are dealing with the same JavaScript objects. Let's delve into the precise problem we're dealing with.

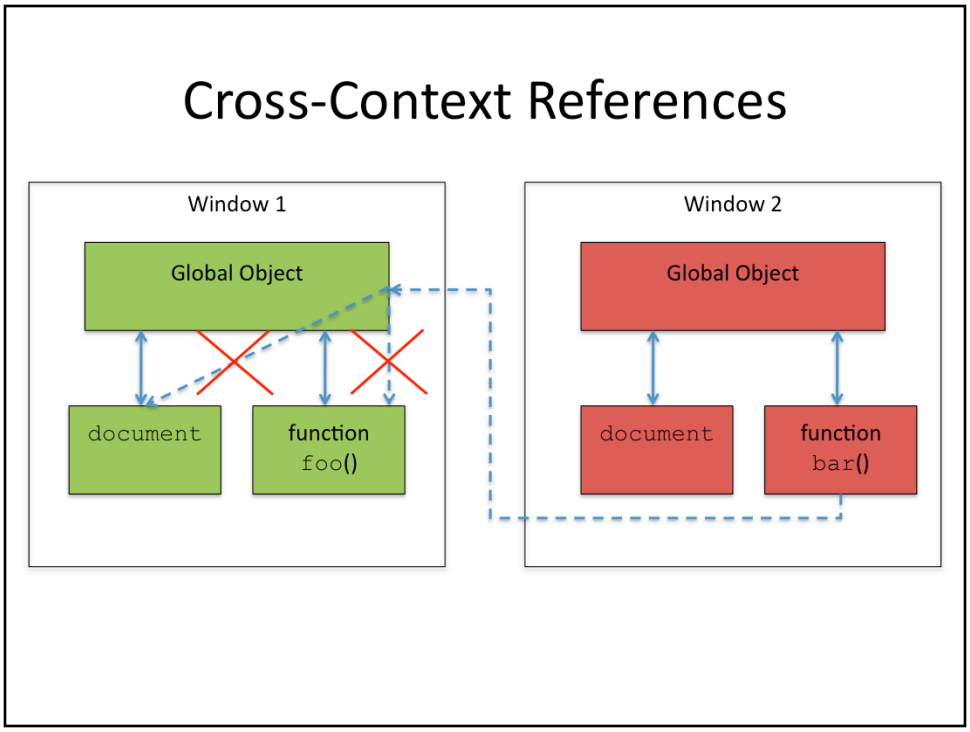


We've been talking about JavaScript rather abstractly so far, but what are all these JavaScript contexts, and what does it mean for a context to reference an object in another context?

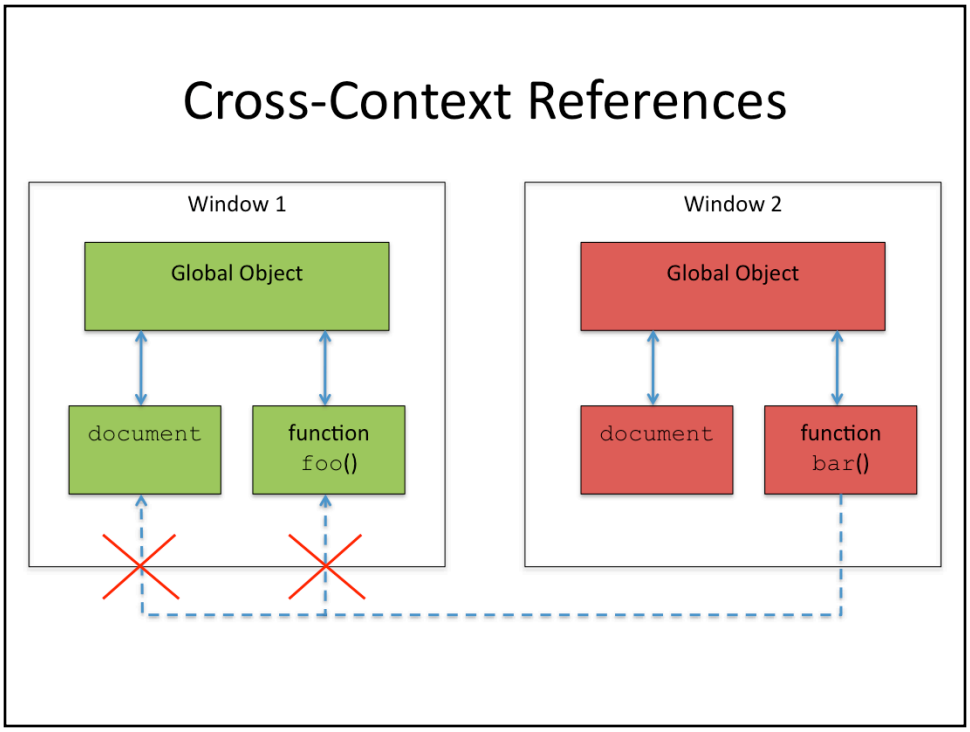
What happens when one context has a reference to an object in another context? It turns out that JavaScript defines a set of very special objects called global objects. Each window and frame has its own global object, and, in fact, JavaScript contexts are defined by JavaScript engines by the global object of the context. Global objects have a number of special properties, the most important of which, for our purposes, is that it is the reference monitor for the DOM discussed earlier. Any context is allowed to access any global object and the it will perform the appropriate access control checks on accessed properties.



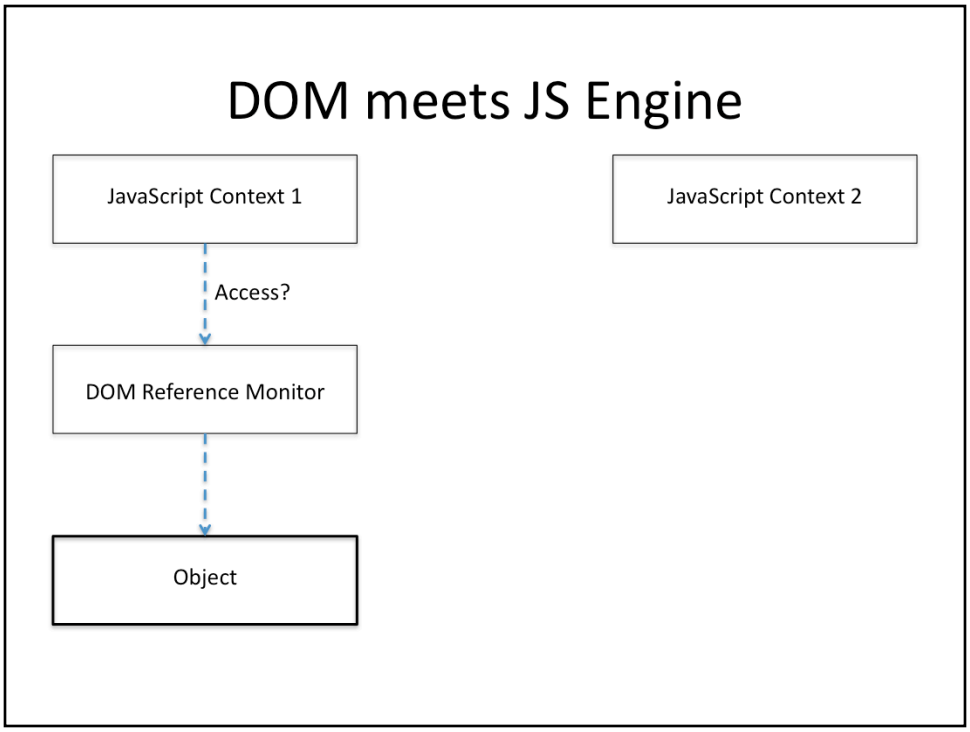
For example, the function “bar” may make a reference to the global object from the context “Window 1.”



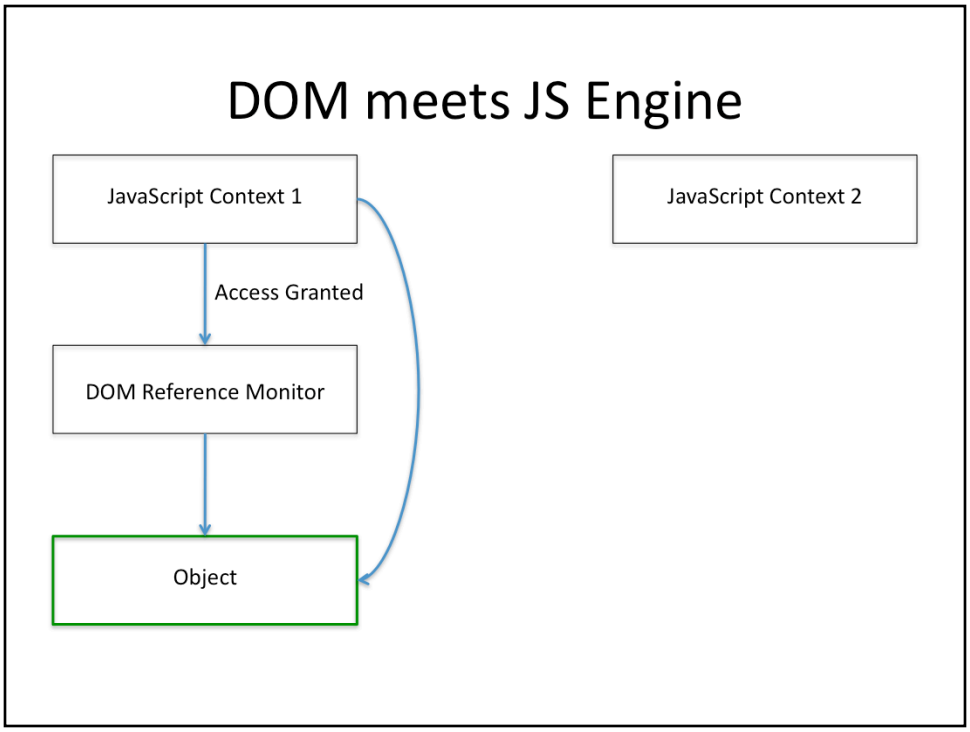
However, it would be bad if bar() was able to reference all of the objects that the global object points to. Fortunately, global objects provide the reference monitor, so this is not an issue.



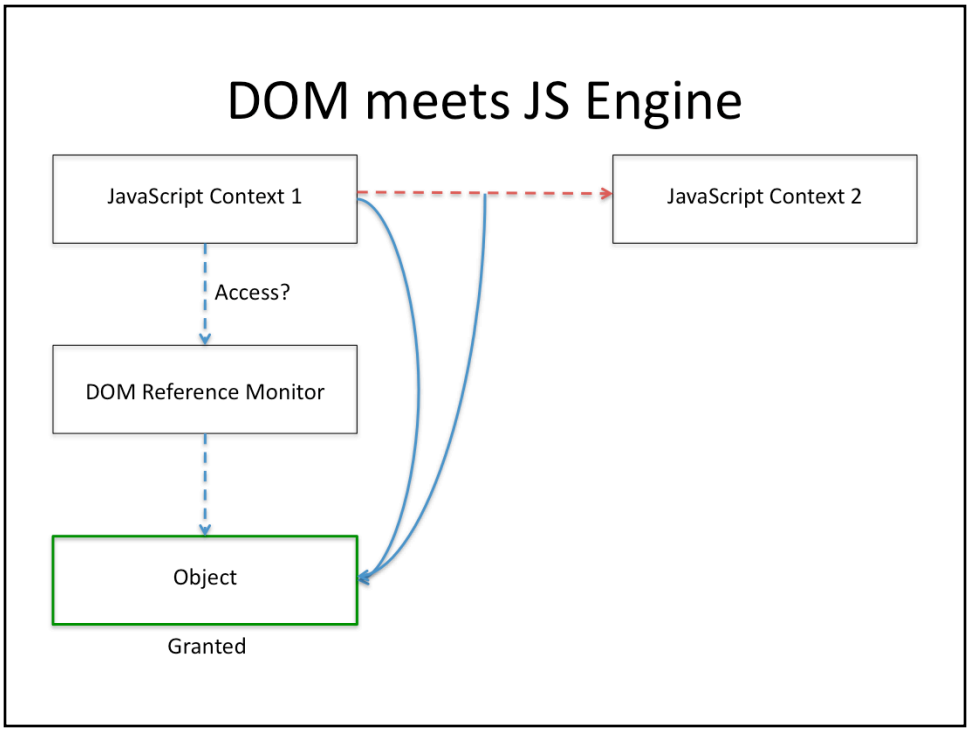
It would also be very bad if bar() held a direct reference to either of the other objects in the “Window 1” context. Unfortunately, they do not have reference monitors wrapping them, so if bar() held a reference to them, it would be game over, unlike if it held a reference to the other context’s global object.



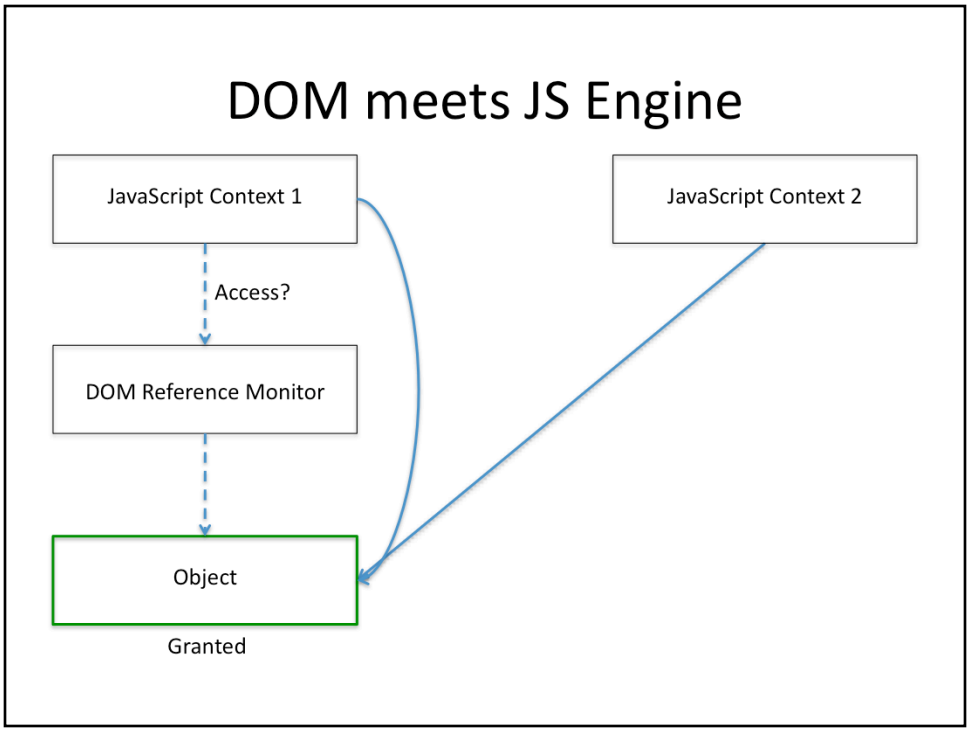
So let's jump back to the two policies of the DOM and JavaScript engine. What happens when the two meet?



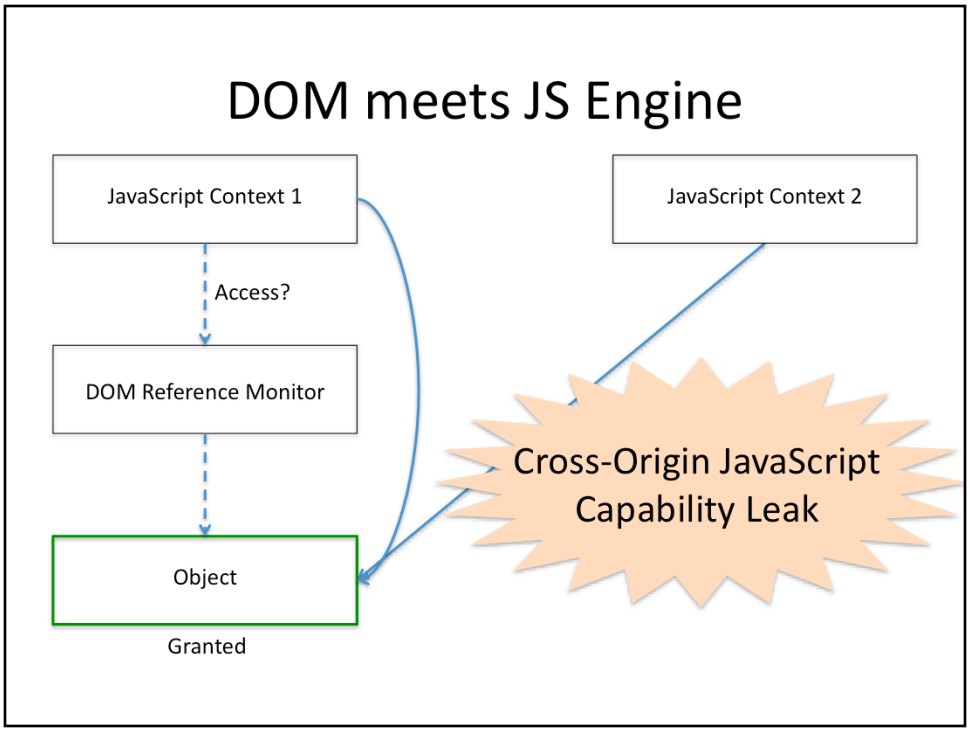
Specifically, let us assume that context 1 is granted access to an object through the reference monitor. From the perspective of the JavaScript engine, the context now holds a reference to the object which is also a capability.



The JavaScript context can do whatever it wants with the reference, including handing the reference to another JavaScript context, on purpose or otherwise.



Because the engine is a capability system, it now can access the object with full permissions. Even though it is a DOM object, it is now bypassing the reference monitor check. Now, we haven't established this a problem yet *per se*; it is not clear that there is any way for a JavaScript context to do this illegitimately. However, it turns out that this is a serious problem because of a number of bugs in web browsers. In these bugs, a malicious script can "trick" the browser into thinking that it's from a different JavaScript context, thus gaining access to a sensitive object through the DOM access control. The malicious JavaScript context now has a capability to this object so it can manipulate it however it sees fit, including all of the things to which it references.

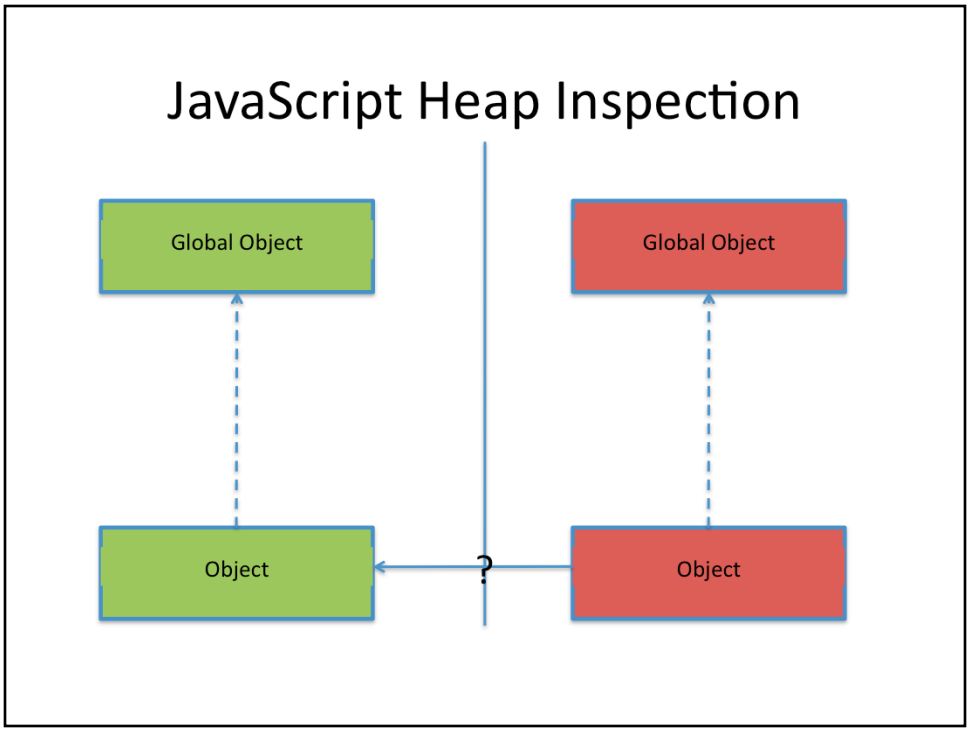


This is a *Cross-Origin JavaScript Capability Leak*. One context *leaks* a capability reference to another context, and this second context now holds an unbridled reference to the DOM object. This is a *very bad thing*.

Overview

- Current JavaScript Security Model
- Cross-Origin JavaScript Capability Leaks
- Capability Leak Detection
- Browser Defense Mechanism

Let's discuss how to help detect these problems in an application using our heap graph analysis tool.

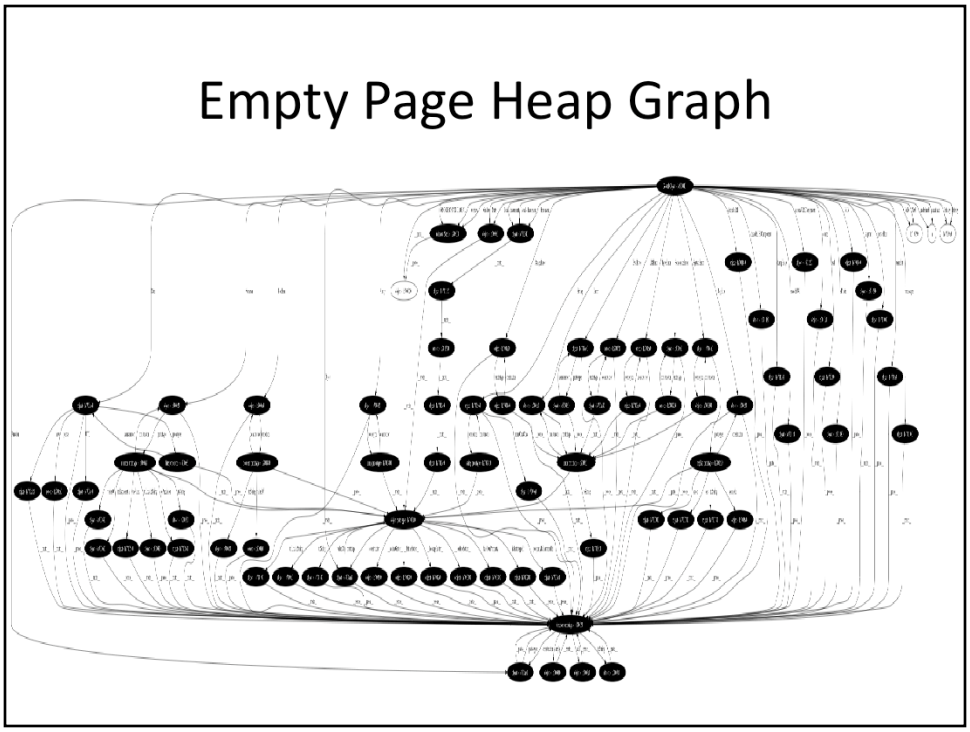


The state we want to detect is when an object from one context holds a reference to an object in a different context. Our solution is to use a heap graph analysis to dynamically mark the JavaScript context of all objects in the JavaScript heap and to through an alert when there is a reference between two objects in different contexts. We modify the WebKit JavaScript engine to perform the instrumentation and analysis for this tool.

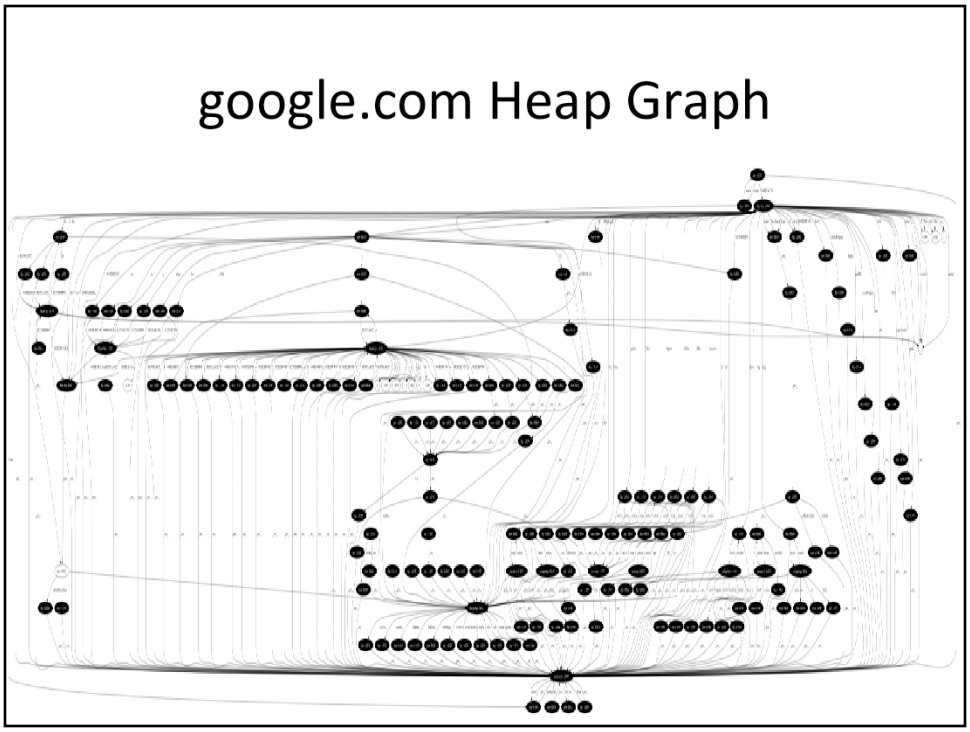
Instrumentation

- In the JavaScript Engine object system
- Object creation, destruction, and reference
- Calls into analysis library

We needed to instrument the WebKit JavaScript engine with calls to our heap graph analysis library. These points are rather straightforward. Rather than putting the instrumentation in the interpreter and JIT, we placed the instrumentation within the object system entirely since that is what we were entirely concerned with. We placed instrumentation points at object creation, object destruction, and the creation of object references (along with several other specialized points).



Here's an graph of the empty page. Because we are tracking all objects on the heap, at any time we can dump an image of the heap as a Graphviz graph. Clearly, even the empty page is rather complex, and these graphs were mainly useful for (a) debugging our work, and (b) reduced versions are useful for finding exploits.

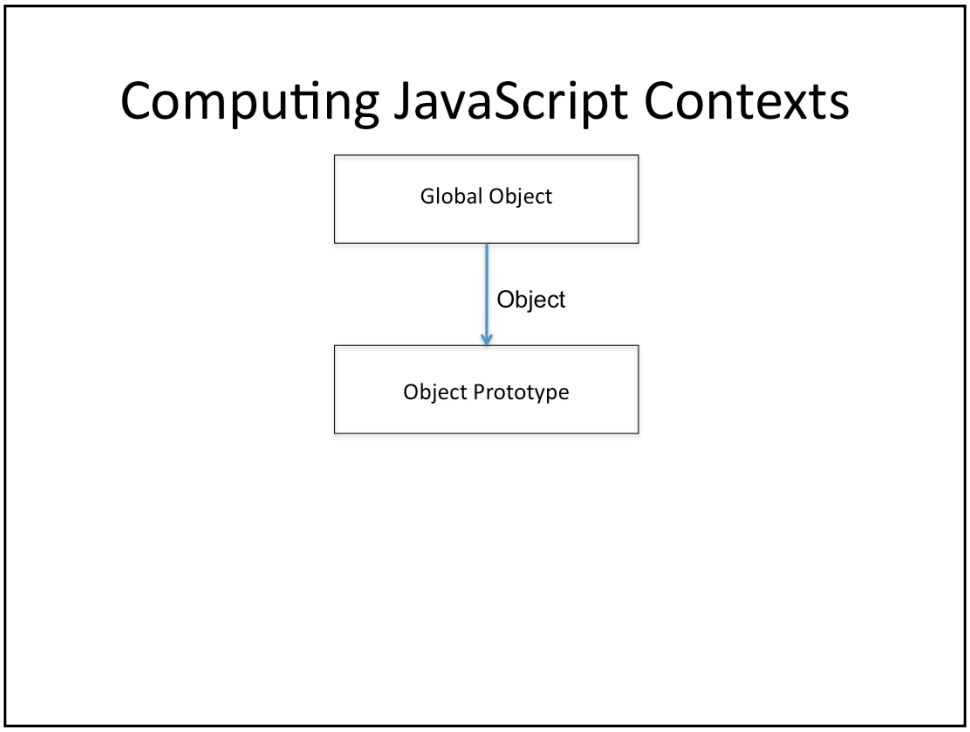


This is the heap graph of google.com. Clearly, more complicated but it turns out that google.com doesn't have that much JavaScript on it and even reaches this level of complexity.

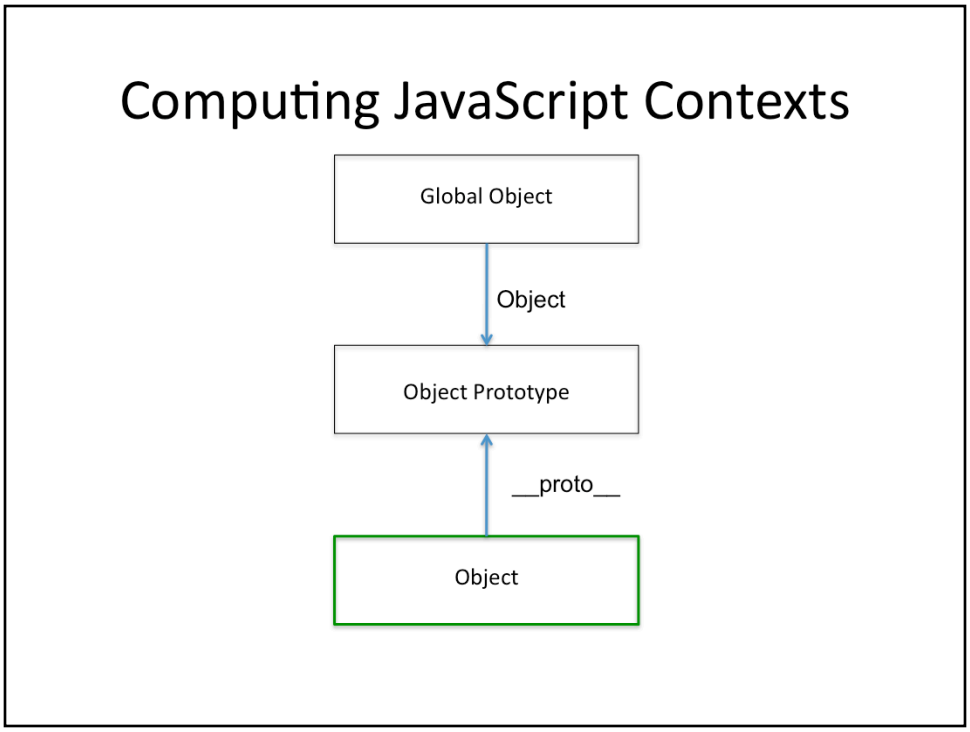
Graph Stats

- empty page
  - 82 nodes
  - 170 edges
- google.com
  - 384 nodes
  - 733 edges
- store.apple.com/us
  - 5332 nodes
  - 11691 edges
- gmail.com
  - 55106 nodes
  - 133567 edges

The graphs can get rather big quickly. While even Google doesn't appear that large, things quickly explode on larger pages, making graphs rather unwieldy. Thus, we realized that we needed to automatically detect violations rather than just manually examining heap graphs.



The key insight to finding these exploits is how the JavaScript context is calculated. Remember that JavaScript contexts are defined by the global object they are associated with. When a new context is created, several things are built, including a instance of a global object, and a unique “object prototype,” which, in the prototype class hierarchy, serves as the ultimate parent of all objects.



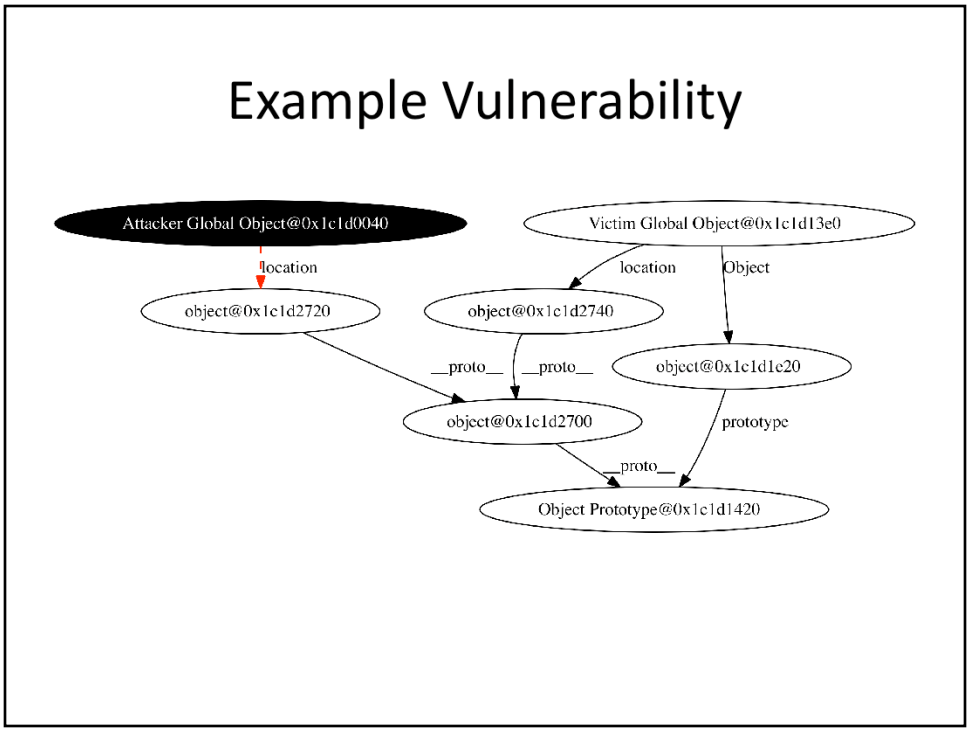
When a new object is created, there is either a direct or indirect path to the Object Prototype. This path goes through the special "__proto__" property. Thus, our algorithm tracks the creation of new contexts, and every time a new object is created, checks the __proto__ property, looking up the referenced object. Because the context is defined by the transitive closure of __proto__ references to the object prototype, we can assign the new object the context of __proto__ object.

Along the way, if we every come across a reference between two objects of different contexts (other than the __proto__ reference), we mark it as a potential problem. Of course, there are some exceptions to this, such as the global object, as discussed earlier, and we white list these.

Generated Coverage

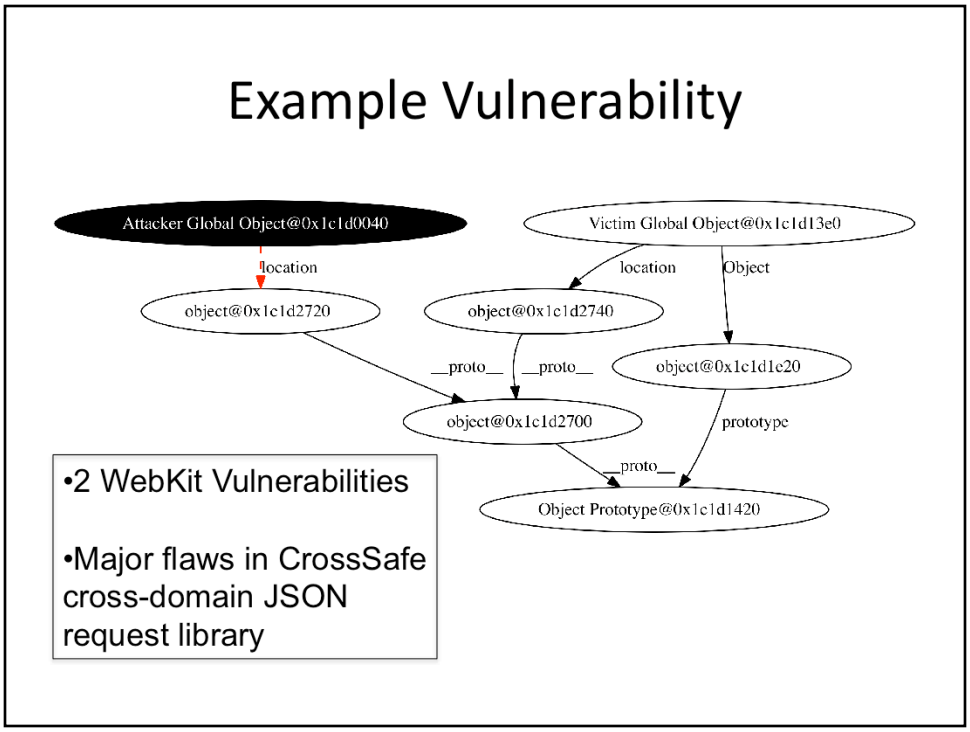
- Total WebKit tests:
  - 9957 tests
- …but most of these tests are for drawing
- Security tests:
  - 143 tests

We were able to generate fairly good coverage by executing our tool across all of the WebKit regression tests. Of course, this is hardly a complete test, but we were simply trying to find proof-of-concept vulnerabilities, not perform an exhaustive search of all possible cross-origin references.



This “zooms in” on one of the vulnerabilities we found. Here, the black represents an object from security context 1 while the white represents objects from security context 2. This is what we particularly want to detect… one JavaScript context referencing another. Despite the graphs being so large, we can perform this reachability analysis rather quickly.

In this example, the vulnerability occurred in WebKit because it was lazily creating the location object. If the location object was created during the execution of another context (i.e. if it belonged to context 1, but context 2 was accessing it), it would be created with the wrong Object prototype. This is dangerous because it allows the object to redefine the behavior of functions, such as toString, that apply to all Objects created in the other context. Then, if that function is called, arbitrary JavaScript will be executed.

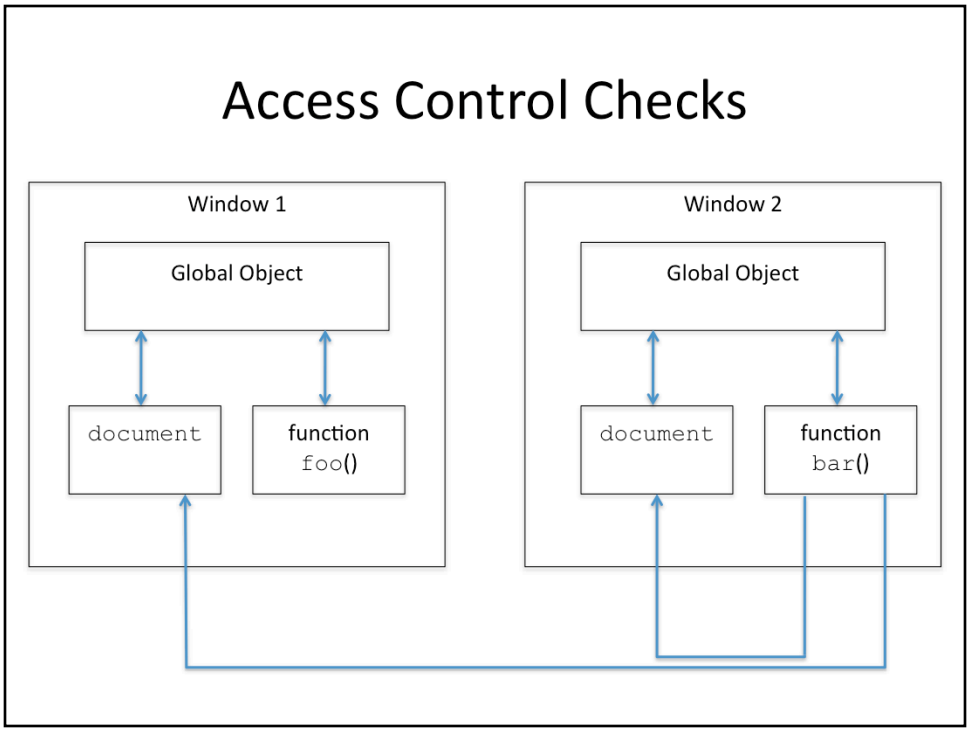


Overall, in our test setup, we found 2 vulnerabilities in WebKit among the 143 tests ran. Additionally we found that the CrossSafe cross-domain JSON request library had a number of vulnerabilities. In all cases, we were able to design subtle exploits of the vulnerabilities that created arbitrary code execution in the other security context.

Overview

- Current JavaScript Security Model
- Cross-Origin JavaScript Capability Leaks
- Capability Leak Detection
- Browser Defense Mechanism

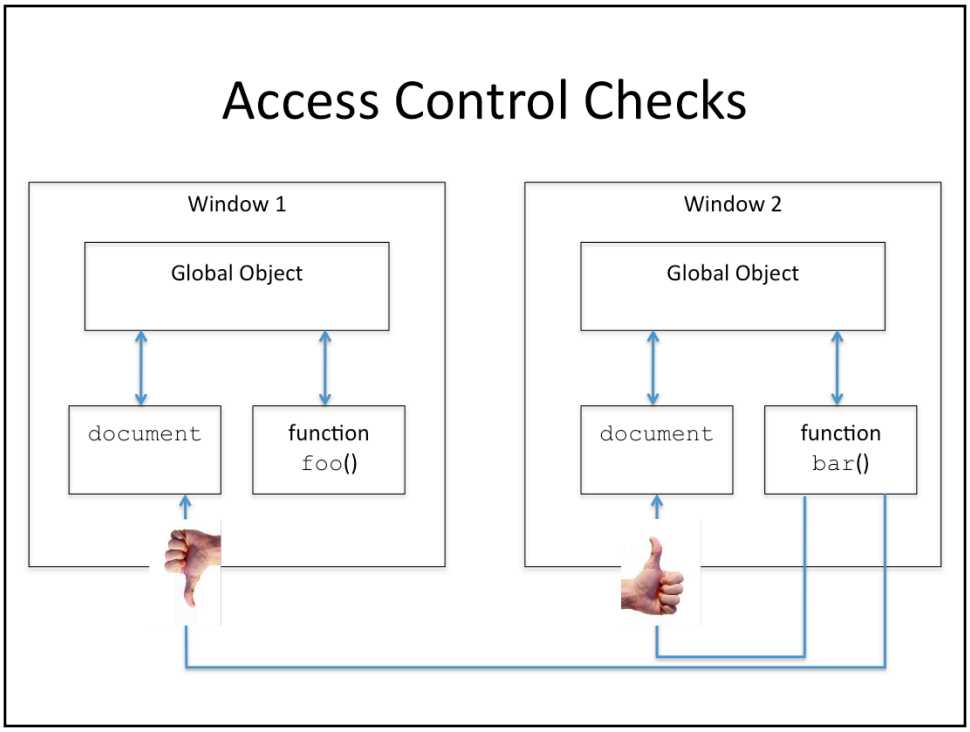
The good news is that we have a proposal to prevent these problems in the future.



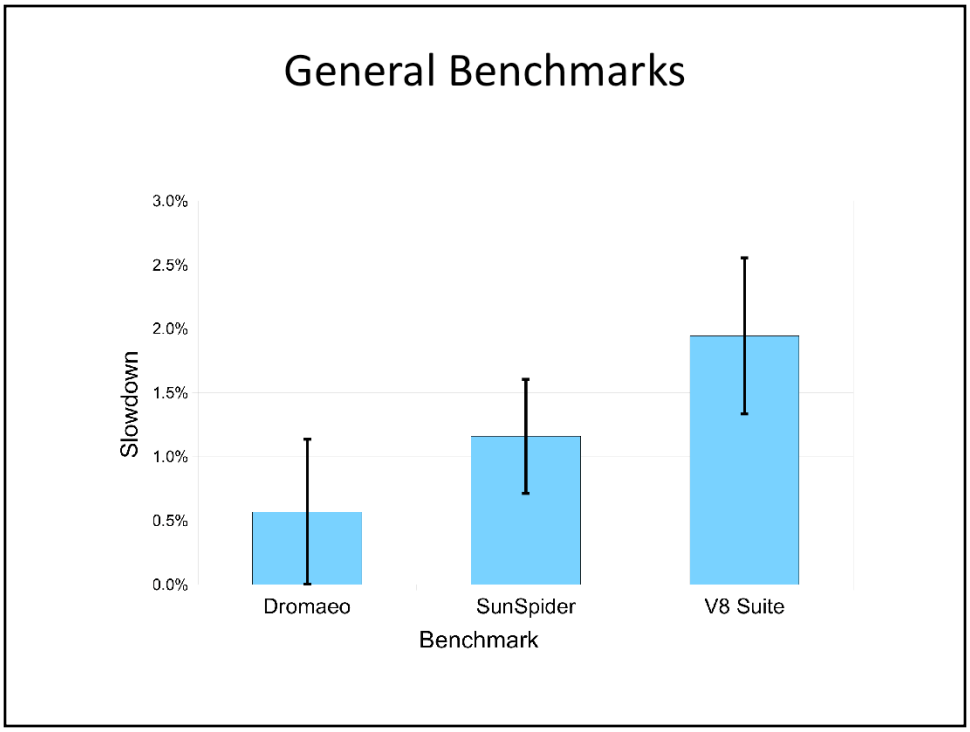
Here we have a small view of some of the objects in current web browsers. For the most part, if there is a leak in the browser that gives an object from context to a second context, that context can access those objects. Yes, there are some exceptions, such as wrapped objects in Firefox, but those are hardly exhaustive and cannot cover cases for which objects are not explicitly wrapped.

In this particular example, function bar() in Window 2 has access to Window 2's document object (as it should), but it also holds a reference to the document object of Window 1, which it can now access.

Our solution is to add an access control check to get and put operations to make it look more like this…

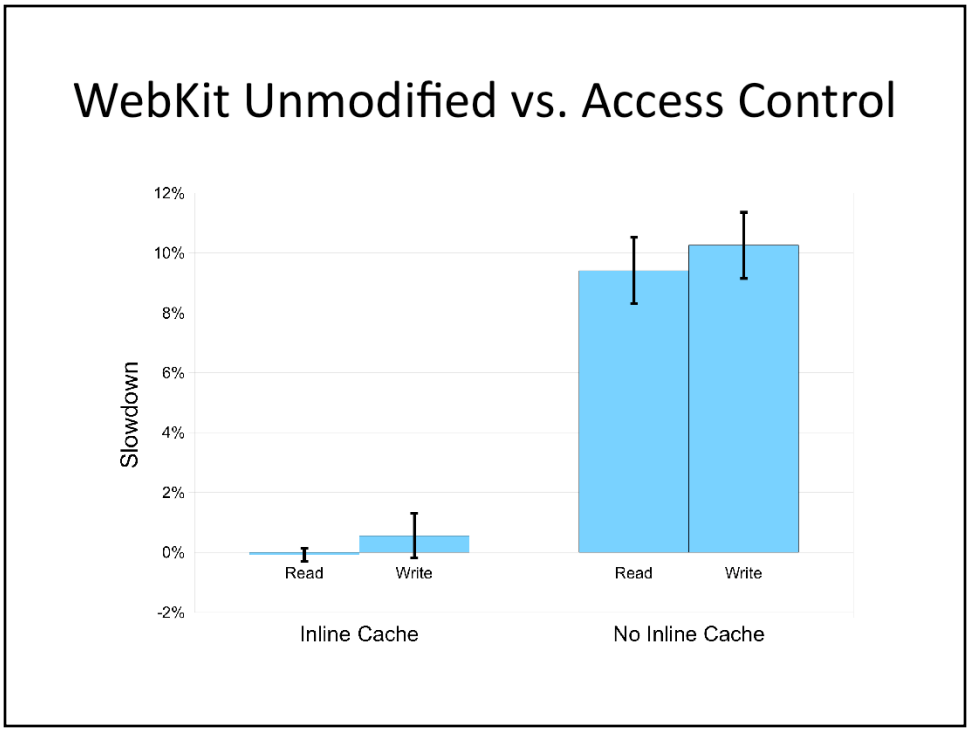


These checks will verify that the JavaScript context of the two objects in question match. If not, the engine should reject the access. It's a simple idea that has been considered in the past. However people have been very concerned about its performance. Additionally, we had concerns initially that it would be difficult to assure that all that places that need to have access control checks would be easy to find and such an implementation would be error prone itself. As to the implementation concerns, we discovered that there are relatively few places that this needs to be actually calculated, and it's fairly clear where those points are. Additionally, in a non-prototype implementation, the access control checks could be built in as a more fundamental and simple mechanism in WebKit, thereby reducing the number of places checks would have to be explicitly placed.



The access control adds negligible performance hits to general benchmarks. Across all of the major industry benchmarks, our access control prototype adds no more than 2% overhead to the base implementation (+/- error).

However, if you consider that In the last year alone there has been a 300% performance increase to WebKit, a 2% hit starts to look a bit paltry..



We hypothesized that our access control was relatively fast because of the inline cache in the new WebKit implementation. In short, for most objects, when a property is looked up the first time, it is looked up in a hash table and the offset into the structure is recorded. When that particular piece of code is accessed again in the future, Instead of hashing in future lookups, the property is accessed by just going directly into the structure with the recorded offset. Because of the offset lookup, we know that the object has access to this object because the first lookup made an access control check. However, whenever a property is deleted, this lookup system is forgone and a hash table lookup is done, making an access control check every time.

In order to test if the inline cache is what's causing the speedup, we made micro-benchmarks for repeatedly reading and writing an object property. In two of the benchmarks, however, we deleted a property from the object first, thus forcing the lookups to occur in the hash table rather than through the inline cache. As the chart clearly shows, where the inline cache is used, there is hardly a noticeable slowdown. However, when the cache is not in use, there is a 9-10% slowdown in the access control implementation.

Conclusion

- Heap Graph Analysis can be used to find vulnerabilities in web browsers
- Web Browsers can provide mechanisms to eliminate these vulnerabilities
- Heap Graph Tool and Access Control Prototype for WebKit:
  - http://webblaze.cs.berkeley.edu/2009/heapgraph

In conclusion, we have introduced a novel tool using heap graph analysis to aid us in finding a new class of vulnerabilities in web browsers, cross-origin JavaScript capability leaks. Additionally, the damage of these vulnerabilities can be mitigated in the future be implementing a new access control mechanism in the web browser.