Talk Notes: Preventing Capability Leaks in Secure JavaScript Subsets

Matthew Finifter
finifter@cs.berkeley.edu

March 3, 2010

Introduction

We are all familiar with online advertising. It's all over the internet. This is the New York Times web site, which, when I counted had 18 ads on the front page. We all see these ads, but have you ever wondered how they get from an advertiser to the website of a content provider and into the browser of an end user? This is the starting point for today's talk.

Overview

I'm going to talk today about ad networks and the threat of malicious advertisements. Then I'll discuss one class of solutions that has been proposed to address this threat. Next, I'll discuss our experiment, which shows how a design choice shared by all current members of this class of solutions provides inadequate security against malicious advertisements. I'll finish by discussing our proposal for fixing this problem.

Ad networks

You have a publisher or a content provider serving juicy content to users. They make advertising revenue by renting out some portion of their web page to one of the many advertising networks out there. Now this ad network can either sell ads directly, or they can rent some of the space they control across various web sites out to some other ad network. This continues until at some point, one of the networks sells the space to an advertiser. The advertiser provides an ad, which travels in the opposite direction to eventually make its way on to the publisher's site and into a user's browser.

So now let's assume that the website doesn't want to serve malware or have a malicious ad launch a cross-site scripting attack against the site. Let's also assume that the advertising networks are not malicious. This may not be true of all advertising networks, but it is certainly true of the largest ones out there. Let's think about how trust flows through this diagram. Without any technical mechanism in place, the publisher trusts the ad network, who trusts each subsequent ad network, who trusts the advertiser. The publisher transitively trusts the advertiser. You may be thinking, so what?

Malicious ads

Well, malicious ads are out there. This is an example of what some users saw a few months ago when they visited the New York Times web site. The ad took over the whole page and attempted to get users to install fake antivirus software. The point is, given the opportunity, malicious advertisers can and will exploit the trust that exists between them and an honest web site.

Safe advertising

Fortunately, there are mechanisms for reducing this trust. There have been several different types of proposals for enabling safe hosting of a guest advertisement on a host publisher's web site.

The first thing that might come to mind is iframes. iframes are supported by all browsers and take advantage of the browser's Same Origin Policy to separate a guest from the host. The problem, though, is that advertisements served in iframes can't be very rich. They can't, for example, grow when the user rolls over them. But advertisers want this kind of rich interaction, and they are willing to pay more to get it. For this reason, it is in the best interest of advertising networks to support this functionality.

Another class of proposals is dynamic enforcers, like for example Caja and Microsoft Web Sandbox. At a basic level, these systems translate guest code into code that can safely be hosted from the same domain as host code. They do this by adding dynamic checks, for example at every property access, to make sure at runtime that the code doesn't do anything that could harm the hosting page. These systems are great, but the added runtime checks can be expensive. This table presents the results of some microbenchmarks that we ran against a few dynamic enforcers. The details of this are in the paper, but the point is that there is significant overhead in the dynamic enforcement approach.

The third approach has been statically verified containment, which I'll focus on for the remainder of the talk.

Statically verified containment

There are 3 serious proposals that fall into this category: ADsafe, Dojo Secure, and Jacaranda.

The idea with statically verified containment is to define a safe language subset for guest code, and to provide a verifier to statically check whether or not a piece of guest code adheres to the subset. If it does, then it will be safely contained when a publisher hosts it. This means that a hosting page can require guest code to be written in the subset before it agrees to host it. Furthermore, each ad network serving as a middleman between the advertiser and the hosting page can also verify an ad's code without changing it.

Common properties

There are some features that the languages in this class share in order to provide statically verified containment of guests running on a host site.

First, they prevent the use of global variables. If an ad, which is supposed to be contained in the page, can get a hold of, say, the document object, then it can control the content of the entire page. I wouldn't consider it contained at that point.

Next, they blacklist dangerous property names. If an ad could access, say, the constructor property of an object, it could redefine the constructors used by the host code. The next time the host constructed a String, for example, it would run whatever code the guest told it to run.

They ban unverifiable constructs, such as eval. Dynamically introduced code cannot be statically verified, which is necessary for the safety properties.

Finally, they provide libraries that guest code can call into to recover some functionality lost by way of some restriction. For example, ADsafe provides an API to replace the bracket operator, which is removed due to the previous restriction, since a blacklisted property name could be constructed at runtime and then accessed via the bracket operator.

Blacklist for property names

Let's consider one of these properties in detail: the blacklisting of dangerous properties. It means that properties known to the designer of the safe subset to be dangerous are disallowed. In other words, it considers only built-in properties – those that exist on every page, including the empty page. If the blacklist

misses one of these, it is insecure on every page because any guest code could use one of the dangerous properties to breach containment. So let's assume that the subset designers did their jobs, and the blacklist is complete for the empty page.

But what about other pages? What other property names can there be, and can they be dangerous?

Well, host code can add properties to built-in objects. For example, it might be convenient for a hosting page to define a right method for Strings that returns the rightmost n characters of the String, and in fact, this is a real example taken from `people.com`. As you might imagine, adding convenience methods like this is a common thing for developers to do.

The problem is that one of these methods could allow a guest to breach containment. Consider what happens if the hosting page defines this code. This code defines an evalMe method that can be called on any String. This method simply calls eval on the value of the String. If the host defines this function, any guest code can call it because a guest can construct a new String and call its evalMe method, which does not appear in the subset designer's blacklist of dangerous property names. In other words, the subset bans the guest from directly using eval because eval would let it breach containment, but now it can indirectly use eval to the same end.

We designed an experiment to understand how common this is in practice. In other words, do existing web sites tend to make properties accessible to guest code that would allow a malicious ad to breach containment? If so, the design choice to use a blacklist of dangerous property names should be reconsidered.

Methodology

In our experiment, we focus on ADsafe as a representative member of this set of language subsets. The set of web sites we looked at is the Alexa US Top 100. We modified the list slightly, for example, by replacing splash pages with pages more representative of those that might host ads. This list of sites was chosen to represent the complexity of JavaScript on popular sites.

For each site, we perform our analysis to see if an ADsafe-verified ad hosted on the site could breach containment to attack the site. We emulate the site hosting ad by using a local HTTP proxy to inject an ad into each site we test.

Browser instrumentation

We instrumented WebKit to perform analysis on each page we load. We record the points-to relation among objects in the JavaScript heap. You can think of this as a graph in which the nodes are objects and an edge represents a property of an object. So if you have code that says a.p=b, we add an edge from the node for a to the node for b.

Suspicious edges

Our analysis compares the set of objects reachable by the guest before and after the host has been loaded on the page. In this diagram, we have some nodes created by the guest code, and some other nodes that exist by default on every page, even the empty page.

We load the guest by itself on the empty page, and mark all nodes transitively reachable from the guest as vetted objects. These are the objects that the guest can access on any page. Again, we assume they are safe because if not, guest code wouldn't be contained on any page. In other words, we assume the subset designer has vetted these objects as safe. This set includes things like the String object, from which you can construct new Strings, and the ADSAFE object, which is ADsafe's support library.

All nodes not marked as vetted in this phase are marked as unvetted by default. This set includes some objects defined by the empty page, such as the document object. This set also includes any nodes added by the host page once it is loaded – these, too, have not been vetted as safe by the subset designer.

Our instrumentation monitors each edge added to the heap graph. If at any point, an edge is added that points from a vetted node to an unvetted node, we flag it. When this happens, it means that some object in the guest code can, through some series of property lookups, have access to an unvetted, and potentially dangerous object. We call such an edge suspicious.

For example, if the host adds an evalMe function to the String prototype, a suspicious edge will be flagged. String.prototype is a vetted node, and the evalMe function is an unvetted node, so this edge in the heap graph is suspicious. What has happened is that the suspicious edge has violated the subset's assumption that all potentially dangerous property names can be statically enumerated.

Suspicious edges

Not all suspicious edges will allow a guest to breach containment. For example, if the host code adds this method to the String prototype, there is no way for guest code to take advantage of it. It can call it, but it doesn't lead to any privileges that the guest doesn't already have. As we've seen though, already, some suspicious edges will allow a guest to breach containment if they give the guest access to something the subset otherwise disallows access to. To find out how many of the sites we tested would allow a guest to breach containment, we manually analyzed each function pointed to by a suspicious edge. We looked in the code of the function for things we knew could let a guest breach containment, like "eval" and "return this."

Results

I've talked about the experiment itself, and now I'll present the results. Of the 90 sites we looked at, 59% contained at least one suspicious edge, and we were able to construct exploits for 37% of the sites. This means that 37% of the sites we tested would be vulnerable to a malicious ADsafe-verified ad if they decided to host such ads.

This number 37% is a lower bound because we may not have detected all suspicious edges, and we may not have found all exploits. Another interesting finding was that exploitability was correlated with the number of suspicious edges. In particular, all but one of the sites with more than 20 suspicious edges were exploitable, and the one that wasn't didn't expose any unvetted methods to the guest. All the unvetted properties it exposed were just data.

Sample exploit

The example that I've been using so far of an exploitable function did not actually appear in any of the sites we analyzed. Something really similar did though. On twitter.com, there is a convenience function for executing all scripts that appear in a string. So you construct a String that contains some HTML, and call its evalScripts method, and all of the scripts in the HTML String get executed. Unfortunately, this means malicious guest code could do the same. It's actually worse than this because this code was not written by twitter. It is part of a JavaScript library called Prototype, which is used by a whole lot of sites. Any of these sites would be vulnerable to the same attack by a malicious ADsafe-verified advertisement.

Possible solutions

Let's step back for a moment and think about the result of the experiment again. We've shown that a substantial portion of the most popular web sites would be vulnerable to a malicious ADsafe-verified advertisement. And the problem is that the host leaves itself vulnerable by adding convenience methods to the prototypes of built-in objects. So it sounds like we have two options here for fixing this problem. The first is to tell publishers to carefully review all their code so that they don't expose any dangerous methods to guest code. And then they can safely use one of these language subsets.

But there are a few problems with this. First of all, this means that each web site has to tailor itself to the restrictions of the particular ad network that it uses. If it is too restrictive, the publisher might simply use a different ad network. Additionally, these problems can be pretty subtle, and they can creep back in at every update to a site. So we don't like this option.

Blancura

Instead, we suggest changing the design of the subset so that these vulnerabilities no longer exist. The fundamental problem is that these subsets use a blacklist to ban specific known-dangerous property names, but they fail to ban property names that become dangerous after being added by the host.

We propose Blancura, which is an extension to ADsafe that uses a whitelist instead of a blacklist for property names. Blancura whitelists property names by running each guest in a separate namespace and requiring that all property names begin with the guest's namespace identifier. We take advantage of the fact that each ADsafe ad is already required to have a unique identifier. As an example of how this works, Blancura code is not allowed to say obj.foo = bar. This code will not pass the verifier. It instead must say obj.BLANCURA_GUEST1_foo = bar. It should be clear that none of the methods previously described as vulnerable will be accessible to guest code.

Blancura

There are some important details to how Blancura works. In particular, you might be thinking, "what about all the useful built-in properties that are safe?" For example, the indexOf function on Strings is pretty useful, and developers might miss it. The Blancura runtime extends the ADsafe runtime to add these properties back in to each guest's namespace like we see up here.

You may also think, "what a pain to have to write the namespace identifier in front of every property name." Well, we thought of that too, so we implemented an idempotent compiler from ADsafe to Blancura. If the compiler sees that the correct namespace identifiers are already being used, it makes no changes to the code.

Finally, we note that Blancura is a strict language subset of ADsafe, so it is at least as safe as ADsafe is, assuming our Blancura runtime does not provide any dangerous objects to guest code.

Conclusion

In conclusion, we focused on statically verified containment, which is one proposal for safely enabling rich advertisements. We designed an experiment to analyze a design choice made by the language subsets that use statically verified containment, and we found that the choice to blacklist dangerous property names is a dangerous ones. A lot of web sites would be left vulnerable to malicious advertisements if they were to host verified advertisements. Finally, we proposed a fix to this problem that uses whitelisting built on separating advertisements into different namespaces. Thank you.