Preventing Capability Leaks in Secure JavaScript Subsets

Matthew Finifter, Joel Weinberger, and Adam Barth
University of California, Berkeley
{finifter, jww, abarth}@cs.berkeley.edu

Abstract

Publishers wish to sandbox third-party advertisements to protect themselves from malicious advertisements. One promising approach, used by ADsafe, Dojo Secure, and Jacaranda, sandboxes advertisements by statically verifying that their JavaScript conforms to a safe subset of the language. These systems blacklist known dangerous properties that would let advertisements escape the sandbox. Unfortunately, this approach does not prevent advertisements from accessing new methods added to the built-in prototype objects by the hosting page. In this paper, we show that one-third of the Alexa US Top 100 web sites would be exploitable by an ADsafe-verified advertisement. We propose an improved statically verified JavaScript subset that whitelists known-safe properties using namespaces. Our approach maintains the expressiveness and performance of static verification while improving security.

1 Introduction

Much of the web economy is fueled by advertising revenue. In a typical advertising scenario, a publisher rents a portion of his or her web page to an advertising network, who, in turn, sublets the advertising space to another advertising network until, eventually, an advertiser purchases the impression. The advertiser then provides content (an advertisement) that the browser displays on the user's screen. Recently, malicious advertisers have sought to exploit these delegated trust relationships to inject malicious advertisements into honest web sites [22, 27]. These attacks are particularly worrying because they undermine confidence in the core of the web economy.

For example, a malicious advertisement exploited The New York Times web site [28] on September 13, 2009. Although most of its advertisements are served via ad networks, The New York Times also includes some advertisements directly from advertisers' servers. In this case, a malicious advertiser posed as Vonage and bought an advertisement. The advertiser then changed the advertisement to take over the entire window and entice users into downloading fake anti-virus software. The New York Times removed the ad but only after a number of visitors were affected.

To protect publishers and end users, advertising networks have been experimenting with various approaches to defending against malicious advertisements. Although there are a wide variety of approaches, ranging from legal remedies to economic incentives, we focus on technological defenses. At its core, displaying third-party advertisements on a publisher's web site is a special case of a mashup. In this paper, we study whether existing techniques are well-suited for containing advertisements and propose improvements to mashup techniques based on static verifiers.

Static verifiers, such as ADsafe [13], Dojo Secure [30], and Jacaranda [15], are a particularly promising mashup technique for advertising. In this approach, the advertising network (and one or more of its syndicates) verifies that the advertisement's JavaScript source code conforms to a particular subset of the JavaScript language with desirable containment properties (see Figure 1). The precise JavaScript subset varies between systems, but the central idea is to restrict the guest advertisement to a well-behaved subset of the language in which the guest can interact only with object references explicitly and intentionally provided to the guest by the host, thus preventing the guest from interfering with the rest of the page.

In this paper, we focus on evaluating and improving the containment of safe JavaScript subsets that use static verification. Static verifiers are appealing because they provide fine-grained control of the advertisement's privileges. For example, the hosting page can restrict the advertisement to instantiating only fully patched versions of Flash Player, preventing a malicious advertisement from exploiting known vulnerabilities in older versions of the plug-in. However, existing static verifiers do not provide perfect containment. To properly contain advertisements, these systems impose restrictions on the *publisher*: the publisher must avoid using certain JavaScript features that let the advertisement breach containment.

These static verifiers impose restrictions on publishers because of a design decision shared by the existing static

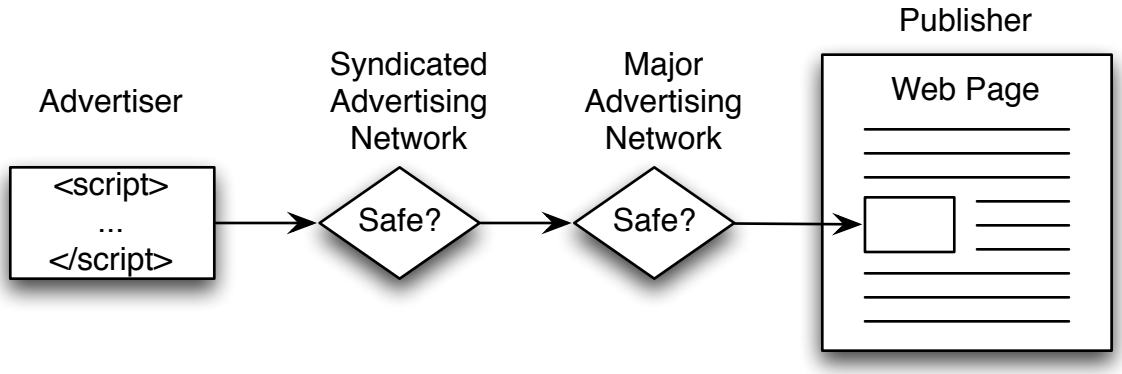


Figure 1. A publisher sells space on his or her page to an advertising network. This space may be resold through multiple advertising networks, until it is sold to an advertiser, who provides an advertisement written in a secure JavaScript subset. The advertisement is checked for safety by each advertising network, in turn, and ultimately served to visitors of the publisher's web page.

verifiers: the verifiers restrict access to object properties using a static blacklist. The blacklist prevents the guest from accessing properties, such as `__proto__`, that can be used to breach containment. The designers of the subsets warrant that their blacklist is sufficient to prevent an advertisement from breaching containment on an otherwise empty page in their supported browsers (or else the subset would always fail to contain advertisements), but the blacklist approach does not restrict access to new properties introduced by the publisher. For example, a publisher might add a `right` method to the string prototype that has not been vetted by the subset designer.

This restriction raises a natural question: how commonly do publishers violate this requirement? If publishers rarely expose new methods (or rarely expose exploitable methods), then this restriction is fairly innocuous. If publishers commonly expose methods that are exploitable, however, then we ought to consider improving these mashup techniques. To answer this question, we designed and implemented an analysis tool that detects which host methods are exposed to guest advertisements. We ran our tool on the non-adult web sites in the Alexa US Top 100 [1] to determine (1) how often sites add methods to prototypes and (2) how many of these methods could be used to breach containment. To measure (1), we rendered the web sites in an instrumented browser that recorded the "points-to" relation among JavaScript objects in the JavaScript heap. Our tool then analyzed the heap and outputted the source code of methods created by the hosting page that would have been exposed to an ADsafe-verified advertisement. To answer (2), we manually analyzed the source code of these methods to determine which sites would have been exploitable by a malicious ADsafe-verified advertisement.

Of the non-adult web sites in the Alexa US Top 100, we found that 59% (53 of 90) exposed new properties to the guest and that 37% (33 of 90) of these sites contained at least one method that could be exploited by an ADsafe-verified advertisement to mount a cross-site scripting (XSS) attack against the publisher. Although the publisher can avoid exposing exploitable methods, even seemingly innocuous methods are exploitable, such as this implementation of `right` found on `people.com`:

```
String.prototype.right = function(n) {
  if (n <= 0) {
    return "";
  } else if (n > String(this).length) {
    return this;
  } else {
    var l = String(this).length;
    return String(this).
        substring(l, l - n);
  }
}
```

We discuss the exploit in detail in Section 4.5. Of the sites with more than 20 exposed properties, only `tagged.com` managed to avoid exposing an exploitable method.

Instead of requiring publishers to vet their exposed methods, we propose eliminating this attack surface by replacing the property blacklist with a whitelist, making static verifiers more robust. Using a technique similar to script accenting [11], we prevent the guest advertisement from accessing properties defined by the hosting page unless those properties are explicitly granted to the guest. Specifically, we restrict the guest to a namespace by requiring all the guest's property names to be prefixed with the guest's page-unique

identifier (which already exists in ADsafe). A guest restricted in this way will not be able to access methods added to the prototype objects by the host because the names of those methods are not in the guest's namespace.

We show that our safe JavaScript subset is as expressive and as easy-to-use as ADsafe by implementing a simple compiler that transforms any ADsafe-verified guest into our proposed subset by prepending the namespace identifier to all property names. Our compiler is *idempotent*: the compiler does not alter code that is already contained in the subset. This property lets us use our compiler as a static verifier for the subset. To check whether a piece of JavaScript code conforms to the subset, one need only run the code through the compiler and check whether the output of the compiler is identical to the input. Idempotency lets each advertising network in the syndication chain apply the compiler to each advertisement without worrying about transforming the advertisement more than once. To protect against a malicious advertising network, the downstream parties (e.g., advertising networks or publishers) can verify each advertisement.

Contributions. We make two main contributions:

- We show that existing static verifiers fall short of defending against malicious advertisements because many publisher web sites use JavaScript features that let the advertisements breach containment.
- We propose a modification these static verifiers that lets publishers host malicious advertisements without requiring the publishers to rewrite their JavaScript.

Organization. The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 details safe JavaScript subsets based on statically verified containment. Section 4 describes our experiment for detecting breaches of containment and reports the results of the experiment. Section 5 proposes Blancura, a safe JavaScript subset based on whitelisting. Section 6 concludes.

2 Related Work

In this section, we survey related work, which falls into two categories: (1) techniques or experiments related to those described herein and (2) other approaches for constructing secure mashups.

2.1 Related Techniques and Experiments

In previous work [10], we used our JavaScript heap analysis framework to find browser bugs that cause cross-origin JavaScript capability leaks and compromise the browser's implementation of the same-origin policy. Our framework consists of two pieces: an instrumented browser that records the "points-to" relation among JavaScript objects in the JavaScript heap and an algorithm for detecting "interesting" edges. In this paper, we reuse the first piece of the framework in our experiment but replace the second piece with a new algorithm that colors nodes based on whether or not they are accessible to an advertisement running by itself on the empty page. In contrast, our previous algorithm detects pointers leaked from one origin to another, which is not applicable in this setting because all the objects that participate in this sort of mashup are contained in the same origin.

In a series of papers [25, 24, 23], Maffeis, Taly, and Mitchell develop a formal semantics for JavaScript and prove the correctness of a specific static verifier (based on FBJS) in their formal model. In the course of writing their proof, they discovered that the host page can compromise its security by extending the built-in prototype objects. They reported this issue to the designers of ADsafe, who added a restriction on the host's behavior to ADsafe's documentation. However, despite recognizing the issue, the language they propose [24] has the same limitations as other blacklist-based static verifiers. We recommend that they adopt our approach and replace their property-name blacklist with a whitelist.

Finally, Yue and Wang study how often web sites use potentially dangerous JavaScript language features, such as `eval` [29]. Similar to our methodology, the authors render the web pages in question using an instrumented browser. Unlike their study, ours is focused on a specific security issue (extending the built-in prototype objects in a way that lets a malicious advertisement breach containment), whereas their study measures a number of general "security hygiene" properties.

2.2 Secure Mashup Designs

Safely displaying third-party advertisements is a special case of the more general *secure mashup problem* in which an integrator (e.g., a publisher) attempts to interact with a collection of untrusted gadgets (e.g., advertisements). The general mashup problem is more difficult than the advertising problem because general mashup techniques aim to provide a high degree of interactivity between the integrator and the gadgets and also among the gadgets themselves.

A number of researchers propose mashup designs based on browser frames (see, for example, [21, 20, 17, 12, 9]). In these designs, the gadget can include arbitrary HTML and JavaScript in a frame. The integrator relies on the browser's security policy to prevent the gadget from interfering in its affairs. In general, these frame-based designs cover a number of use cases, including advertising, but they have a coarse-grained privilege model. For example, HTML5's `sandbox` attribute [6] can restrict the frame's privileges

in a number of enumerated ways, but language-based designs can grant finer-grained privileges not pre-ordained by browser vendors. Specifically, a language-based mashup can restrict the gadget to certain (e.g., fully patched) versions of specific plug-ins (e.g., Flash Player), whereas the `sandbox` attribute is limited to allowing or denying all plug-ins.

Gatekeeper [19] is a "mostly static" enforcement mechanism designed to support a rich policy language for complex JavaScript widgets. Although mostly a static analyzer, Gatekeeper uses runtime enforcement to restrict some JavaScript features, such as `eval`. To confine widgets, Gatekeeper analyzes the whole JavaScript program, including the page hosting the widget. This use of whole program analysis makes Gatekeeper an awkward mechanism for restricting the privileges of advertisements because the advertising network typically does not have the entire source code of every publisher's web site.

Like the static verifiers we study in this paper, dynamic enforcers, such as Caja [2] and Web Sandbox [3], restrict gadget authors to a subset of the JavaScript language with easier-to-analyze semantics. Unlike static verifiers, dynamic enforcers transform the source program by inserting a number of run-time security checks. For example, Caja adds a dynamic security check to every property access. These dynamic enforcement points give these designs more precise control over the gadget's privileges, typically letting the mashup designer whitelist known-good property names.

Inserting dynamic access checks for every property access has a runtime performance cost. To evaluate the performance overhead of these dynamic access checks, we used the "read" and "write" micro-benchmarks from our previous study [10], which test property access time. We ran the benchmarks natively, translated by each of Caja's two translators (Cajita and Valija), and translated by Web Sandbox. We modified the existing benchmarks in three ways:

1. We modified our use of `setTimeout` between runs to conform to the JavaScript subset under test by using a function instead of a string as the first argument.

2. Instead of running the benchmark in the global scope, we ran the benchmark in a local scope created by a closure. This improved the Valija benchmark enormously because accessing global variables in Valija is extremely expensive.

3. For Cajita and Valija, we reduced the number of iterations from one billion to one million so that the benchmark would finish in a reasonable amount of time. For Microsoft Web Sandbox, we reduced the number of iterations from one billion to 10,000 because the Microsoft Web Sandbox test bed appears to have an execution time limit.

	read	write
Cajita	21%	20%
Valija	1493%	1000%
Microsoft Web Sandbox	1217%	634%

Table 1. Slowdown on the "read" and "write" micro-benchmarks, average of 10 runs.

We ran the benchmarks in Firefox 3.5 on Mac OS X using the Caja Testbed [5] and the Microsoft Web Sandbox Testbed [26].

Our findings are summarized in Table 1. The Cajita translation, which accepts a smaller subset of JavaScript, slows down the two micro-benchmarks by approximately 20%. Even though Cajita adds an access check to every property access, the translator inlines the access check at every property access, optimizing away an expensive function call at the expense of code size. Valija, which accepts a larger subset of JavaScript, slows down the read and write micro-benchmarks by 1493% and 1000%, respectively. Valija translates every property access into a JavaScript function call that performs the access check, preventing the JavaScript engine from fully optimizing property accesses in its just-in-time compiled native code. Microsoft Web Sandbox performs similarly to Valija because its translator also introduces a function call for every property access. More information about Caja performance is available on the Caja web site [7].

Dynamic enforcers are better suited for the generic mashup problem than for the advertising use case specifically. In particular, advertising networks tend to syndicate a portion of their business to third-party advertising networks. When displaying an advertisement from a syndication partner, an advertising network must verify that the advertisement has been properly transformed, but verifying that a dynamic enforcer's transformation has been performed correctly is quite challenging. By contrast, static verifiers are well-suited to this task because each advertising network in the syndication chain can run the same static verification algorithm.

3 Statically Verified Containment

One popular approach to designing a secure JavaScript subset is to *statically verify containment*. This approach, used by ADsafe [13], Dojo Secure [30], and Jacaranda [15], verifies that the contained code is in a well-behaved subset of the JavaScript language. In this section, we describe the characteristics and limitations of the current languages that use this approach.

One important use of these languages is to prevent "guest" code (e.g., advertisements) from interfering with

the "host" web page running the guest code. The guest script should be contained by the language to run as if it were run by itself on an otherwise empty page. ADsafe is specifically intended for this use [13], and proponents of Dojo Secure and Jacaranda also envision these languages used for this purpose [16, 30].

3.1 Containment Architecture

Secure JavaScript subsets that use statically verified containment prevent guests from using three classes of language features, described below. If left unchecked, a guest could use these language features to escalate its privileges and interfere with the host page. The details of these restrictions vary from language to language. For the precise details, please refer to the specifications of ADsafe, Dojo Secure, and Jacaranda.

- **Global Variables.** These languages prevent the guest script from reading or writing global variables. In particular, these languages require that all variables are declared before they are used (to prevent unbound variables from referring to the global scope) and forbid the guest from obtaining a pointer to the global object (to prevent the guest from accessing global variables as properties of the global object). For example, these languages ban the `this` keyword, which can refer to the global object in some contexts.

- **Dangerous Properties.** Even without access to global variables, the guest script might be able to interfere with the host page using a number of "special" properties of objects. For example, if the guest script were able to access the `constructor` property of objects, the guest could manipulate the constructors used by the host page. The languages implement this restriction by blacklisting a set of known-dangerous property names.

- **Unverifiable Constructs.** Because dynamically generated scripts cannot be verified statically, these languages also ban language constructs, such as `eval`, that run dynamic script. In addition to dynamic code, the languages also ban dynamic property access via the subscript operator (e.g., `foo[bar]`) because `bar` might contain a dangerous property name at run time. These languages typically do allow dynamic property access via a library call, letting the library check whether the property being accessed at runtime is on the blacklist, and if not, allow the access.

These languages enforce the above-mentioned restrictions using a static *verifier*. The verifier examines the source code of the guest script and checks whether the script adheres to the language's syntactic restrictions. In typical deployments, these languages provide the guest script a library for interacting with the host page. For example, ADsafe provides widgets with a jQuery-style library for accessing a subset of the hosting page's Document Object Model (DOM). These libraries typically interpose themselves between the guest script and the host objects, preventing the guest from interacting with the host in harmful ways. For example, the library might restrict the guest to part of the DOM by blocking access to the `parentNode`. Even in languages that statically verify containment, these libraries often involve dynamic access checks.

3.2 Limitations

ADsafe, Dojo Secure, and Jacaranda all block access to dangerous properties using a blacklist. When analyzing a guest script, the static verifier ensures that the script does not use these specific property names. This approach works well on the empty page where the set of property names is known (at least for a fixed set of browsers), but this approach breaks down if the host inadvertently adds dangerous properties to built-in object prototypes because these new properties will not be on the verifier's blacklist [24].

When the host page extends the built-in object prototypes by adding new properties, those properties are visible on all objects from that page. For example, if the host page added the function `right` to `String.prototype`, then the `right` property would be visible on all strings because the JavaScript virtual machine uses the following algorithm to look up the value of a property of an object:

1. Check whether the property exists on the object itself. If so, return the value of the property.

2. Otherwise, continue searching for the property on the object's prototype (identified by the `__proto__` property), returning the value of the property if found.

3. If the object does not have a prototype (e.g., because it's the root of the prototype tree), then return `undefined`.

All the prototype chains in a given page terminate in the `Object.prototype` object. If the host page adds a property to `Object.prototype`, the property will appear on all objects in the page. Similarly, adding a property to `String.prototype` or `Array.prototype` causes the property to appear on all strings or arrays (respectively).

For this reason, ADsafe explicitly admonishes host pages not to extend built-in prototypes in dangerous ways:

> None of the prototypes of the builtin types may be augmented with methods that can breech [sic] ADsafe's containment. [13]

However, ADsafe also says that "ADsafe makes it safe to put guest code (such as third party scripted advertising or widgets) on any web page" [13]. These two statements appear to be in conflict if web sites commonly augment built-in types with methods that breach ADsafe's containment.

4 Detecting Containment Breaches

In this section, we evaluate whether existing static verifiers can be used to sandbox advertisements on popular web sites. We automatically detect "unvetted" functions exposed to guest scripts and then examine these function to determine whether they introduce vulnerabilities that could be exploited if the page hosted ADsafe-verified ads.

4.1 Approach

To detect when objects created by the host page are accessible to the guest, we load the host page in an instrumented browser. The instrumented browser monitors JavaScript objects as they are created by the JavaScript virtual machine. Our instrumentation also records the "points-to" relation among JavaScript objects. Whenever one object is stored as a property of another object, we record that edge in a graph of the JavaScript heap.

To determine whether an object created by the host page is accessible to the guest, we compare the set of JavaScript objects accessible to the guest script on the empty page with the set accessible to the guest script on the host page. We call an edge from an object in the first set to an object in the second set *suspicious*. The designers of the secure JavaScript subset vet the objects accessible to the guest on the empty page to ensure that guest code cannot breach containment using those objects (or else the subsets would fail to be secure on virtually every page), but the guest script might still be able to leverage suspicious edges to escape the sandbox.

Not all suspicious edges are exploitable. For example, if the host exposes only the identity function to the guest, the guest cannot leverage this function to escalate its privileges. However, if the host defines a function that calls `eval` on its argument and exposes that function to the guest script, then the guest script can compromise the host completely. To determine whether a suspicious edge lets a guest breach containment, we examined the source code of newly accessible functions. If we are able to construct a proof-of-concept guest script that exploits such a function, we say that the host page has *violated* the containment policy.

4.2 Design

To compute the set of suspicious edges, we classify objects in the host page's JavaScript heap into two sets:

- **Vetted.** The *vetted objects* are the JavaScript objects accessible by the guest script on the empty page. In particular, the objects created by the guest script itself are vetted, as are objects reachable from those objects on the empty page. We rely on the designer of the secure JavaScript subset to ensure that these objects are safe for the guest script to access.

- **Unvetted.** The *unvetted objects* are the JavaScript objects that are not accessible by the guest script on the empty page. For example, all the objects created by the host page are unvetted because those objects are not present on the empty page. As another example, the `document` object is unvetted because it is not accessible by the guest script on the empty page, even though `document` exists on the empty page.

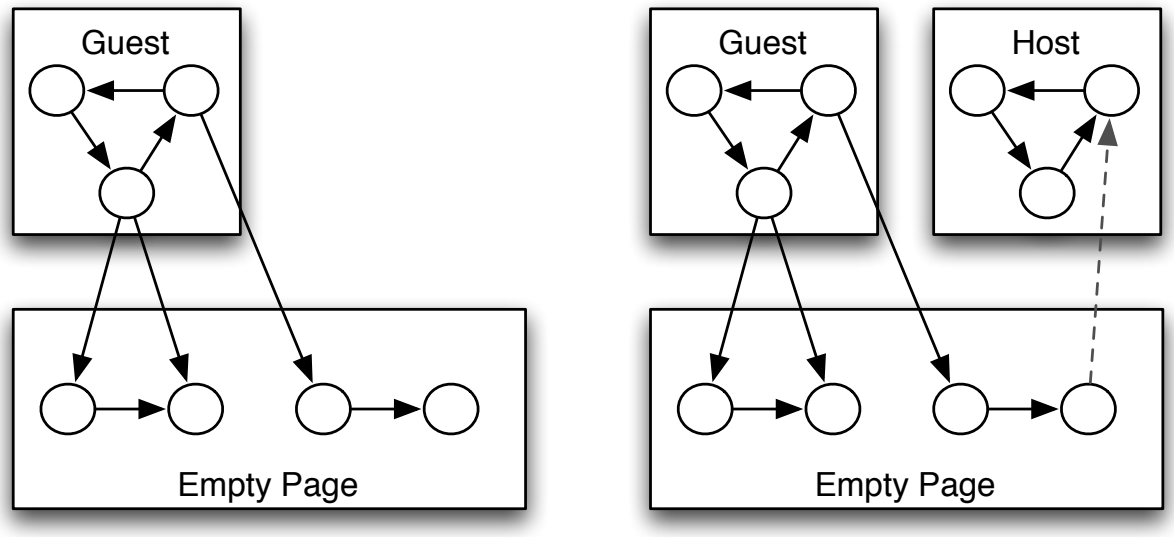
We use a *maximal guest* to detect the set of vetted objects. The maximal guest accesses the set of built-in objects allowed by the rules of the safe JavaScript subset.[1] Note that creating multiple instances of these objects does not expand the set of built-in objects reachable by the guest.

We classify the objects in the JavaScript heap for a given page as vetted or unvetted using a two phase algorithm. We load the page in question via a proxy that modifies the page before it is rendered by an instrumented browser.

- **Phase 1:** Before loading the page's HTML, the proxy injects a script tag containing the maximal guest. The instrumented browser seeds the set of vetted objects by marking the objects created by the maximal guest as vetted. We expand the set to its transitive closure by crawling the "points-to" relation among JavaScript objects in the heap and marking each visited object as vetted (with some exceptions noted below). Just prior to completing, the maximal guest calls a custom API to conclude the first phase.

- **Phase 2:** After the maximal guest finishes executing, all the remaining JavaScript objects in the heap are marked as unvetted. As the instrumented browser parses the host page's HTML and runs the host's scripts, the newly created objects are also marked as unvetted. Whenever the instrumented browser detects that the host page is adding an unvetted object as a property of a vetted object, the browser marks the

[1] For ADsafe, the maximal guest we constructed instantiates one each of the following object types: `Function`, `Object`, `Array`, `RegExp`, `String`, `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. The only built-in objects defined by WebKit but absent from this list are `Number`, `Boolean`, and `Date`. `Date` objects are disallowed in ADsafe code. Numeric and Boolean literals (but not objects) can be constructed by ADsafe code, but these were not included in our maximal guest due to an implementation error. Because of this and other factors, our result regarding the fraction of exploitable sites is a lower bound.



(a) Before loading host code

(b) During loading of host code

Figure 2. Depiction of a simple heap graph. In (a), all nodes are vetted nodes. In (b), a suspicious edge has been detected. The host code has added a pointer that provides the guest code with access to an unvetted object. The suspicious edge points from the prototype of a built-in object to a method defined by the host.

corresponding edge in the heap graph as suspicious and records the subgraph reachable via this suspicious edge. The dashed line in Figure 2 represents such a suspicious edge.

One difficulty is knowing when to end the second phase. In principle, a web page could continue to add suspicious edges indefinitely. In practice, however, most of the suspicious edges we found were added by "library code" that is run while the page is loading so that most scripts in the page can make use of the extensions provided by the library. To terminate Phase 2, the proxy inserts a script tag after the page's HTML that calls a custom API.

When computing the set of vetted objects as the transitive closure of the initial vetted objects, the instrumented browser considers only edges in the "points-to" relation that can be traversed by guest scripts in the safe subset. For example, in ADsafe, guest scripts cannot access properties with certain names, such as `arguments`, `constructor`, and `eval`. Additionally, ADsafe guests cannot access any properties with names that begin with an underscore character. These properties are ignored in computing the transitive closure, with one notable exception: `__proto__`. Although the guest script cannot access `__proto__` explicitly, the guest script can access the property *implicitly* because this property is used by the algorithm described in Section 3.2 for resolving property names.

4.3 Implementation

We implemented our instrumented browser by modifying WebKit, the open-source browser engine that powers Safari and Google Chrome. Our work builds off of our previous JavaScript heap inspection implementation [10]. We implemented our HTTP proxy in Perl using the `HTTP::Proxy` library. The proxy modifies `text/html` HTTP responses in flight to the browser. Because the browser can parse malformed HTML documents with a `<script>` before the main `<html>` and after the main `</html>`, we did not need to modify the response except to prepend and to append the required script blocks.

4.4 Experiment

To evaluate whether web sites commonly use JavaScript features that breach the containment of existing statically verified JavaScript subsets, we analyzed the Alexa US Top 100 web sites using our breach detector for ADsafe. This set of web sites was chosen to represent the level of complexity of JavaScript code found on popular web sites. Although this set might not be representative of all web sites, malicious advertisements on these web sites impact a large number of users.

4.4.1 Methodology

We retrieved the list of Alexa US Top 100 web sites and tested those web sites in May 2009. Instead of using the home page of every site on the list, we made the following modifications:

- We removed all 9 web sites with adult content. (We were unsure whether our university permits us to browse to those web pages in our workplace.)
- We removed `blogspot.com` because its home page redirected to `blogger.com`, which was also on the list. The `blogspot.com` domain hosts a large number of blogs, but the Alexa list did not indicate which blog was the most popular, and we were unsure how to determine a "representative" blog.
- For some web sites, the home page is merely a "splash page." For example, the `facebook.com` home page is not representative of the sorts of pages on Facebook that contain advertisements. Using our judgment, we replaced such splash pages with more representative pages from the site. For example, on `facebook.com`, we chose the first page presented to the user after login.

We then used the algorithm described above to detect suspicious edges in the JavaScript heap graph of the main frame created by visiting each of the 90 web pages we selected for our experiment. We observed the number of suspicious edges for each page as well as the source code of the exposed methods. We then manually analyzed the source code of the exposed methods to determine whether the suspicious edges would have been exploitable had the selected web pages hosted ADsafe-verified advertisements. Upon finding one exploitable method for a given site, we stopped our manual analysis of the remaining exposed methods for that site because an attacker needs only one vulnerability to construct an exploit.

One potentially fruitful area of future work is to automate this manual analysis step to scale our analysis technique to a larger population of web sites (such as the entire web). In fact, we used a crude static analysis framework based on regular expressions (e.g., `grep` for `return\s+this` and `eval`) to help us find vulnerabilities faster. Even with these simple tools, the manual analysis required only a number of hours.

4.4.2 Results

In our experiment, we observed the following:

- Of the web pages, 59% (53 of 90) contained at least one suspicious edge.
- We were able to construct proof-of-concept exploits for 37% (33 of 90) of the web pages.

Figure 3 summarizes our results. We observed a max of 72 suspicious edges and a mean of 16.2 ($n = 90$, $\sigma = 23.8$).

4.4.3 Discussion

Our experiment lower bounds the number of analyzed sites that could be exploitable by a malicious ADsafe-verified advertisement because we might not have found every suspicious edge or every exploit. The histogram in Figure 3 shows that the number of suspicious edges is correlated with exploitability. Although not all functions are exploitable (and not all suspicious edges lead to functions), many common JavaScript programming idioms (such as returning `this`) are exploitable. Of the sites with 20 or more suspicious edges, all but one violate containment. The one site that did not violate containment, `tagged.com`, did not contain any unvetted functions.

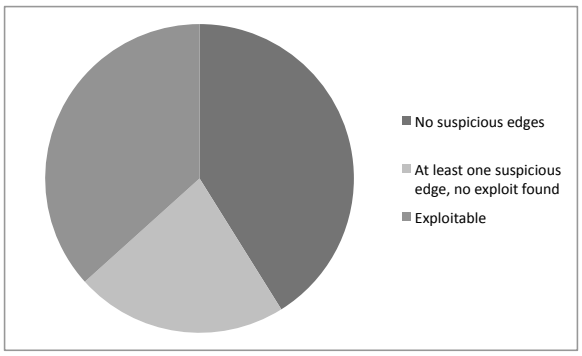
4.5 Case Studies

We illustrate how a malicious guest can use functions supplied by the host page to compromise the security of the sandbox by examining two case studies from popular web sites. In all the cases we examined, when a guest could escalate its privileges using host-supplied functions, the guest could run arbitrary script with the authority of the hosting page, completely breaching containment.

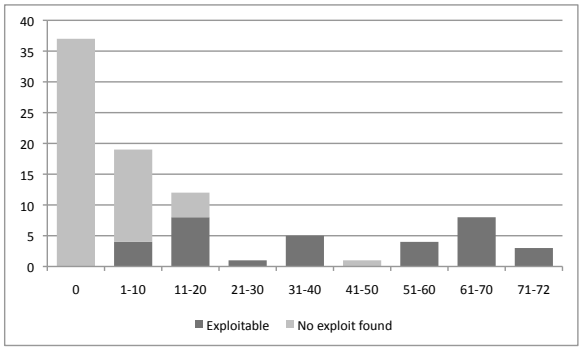
4.5.1 People

We first examine `people.com`, a magazine. People augments the `String` prototype object with a number of utility methods, including a method `right` that extracts the n right-most characters of the string. As an optimization, the function returns the original string (designated by `this`) if it contains fewer than n characters. Although apparently benign, this optimization lets a guest breach containment.

When a JavaScript function is called as a method of an object (e.g., as in `obj.f()`), the `this` keyword refers to that object (e.g., `obj`). However, if a function is called in the global scope (e.g., as in `f()`), then the `this` keyword refers to the global object, commonly known as the `window` object. By removing the `right` function from a primitive string, as shown in Figure 4, a malicious guest can call the function in the global scope, binding `this` to the global object. By supplying a sufficiently large value for `n`, the host's function will return the global object to the attacker. Once the attacker has access to the global object, the attacker can use any number of methods for running arbitrary script with the host's authority.



(a) Percentage of sites with no suspicious edges, those with suspicious edges that we were not able to exploit, and those that we were able to exploit.



(b) Histogram depicting the number of sites for different ranges of numbers of violations, as well as the fraction of those for which at least one exploit was found.

Figure 3. Visual depictions of the results of the experiment.

```
String.prototype.right = function(n) {
  if (n <= 0) {
    return "";
  } else if (n > String(this).length) {
    return this;
  } else {
    var l = String(this).length;
    return String(this).substring(l, l - n);
  }
}
```

(a) Relevant host code

```
<div id="GUESTAD_">
<script>
"use strict";
ADSAFE.go("GUESTAD_", function (dom, lib) {
  var f = "Hello".right;
  f(100).setTimeout("... attack code ...", 0);
});
</script>
</div>
```

(b) ADsafe-verified guest code implementing exploit

Figure 4. Exploit for `people.com`

4.5.2 Twitter

Next, we examine `twitter.com`, a social network. Twitter makes use of the Prototype JavaScript library [8], which adds a number of useful functions to the default JavaScript environment. The exploit we describe for Twitter is applicable to every web site that uses the Prototype Library, including numerous prominent sites [4]. Moreover, we argue that the particular function we exploit in the Prototype Library is not an isolated example because the primary purpose of the Prototype Library is to augment the built-in prototype objects with useful (but potentially exploitable) functions.

The Prototype Library adds a method to the `String` prototype that calls `eval` on the contents of every `<script>` tag found in the string. Exploiting this function is trivial: a malicious advertisement need only create a string containing a `<script>` tag and call its `evalScripts` method (see Figure 5). This example illustrates that library designers do not expect their functions to be called with untrusted arguments. This trust assumption is currently valid, as evidenced by the lack of actual exploits against Twitter, but becomes invalid when hosting ADsafe-verified advertisements.

To protect themselves from this attack vector, web sites could remove all potentially exploitable functions, but this approach has two serious drawbacks. First, the publisher must modify its page to the specifications of the advertising network. If an advertising network imposes restrictions on how their publishers code their web sites, publishers are likely to sell their advertising space to another advertising network rather than retrofit their web sites. Second, JavaScript contains many subtly dangerous idioms. Even if a publisher can rid every page of these idioms, the idioms are likely to return as developers modify the site. Instead of forcing each publisher to modify their web pages, we suggest strengthening the sandbox to prevent these prototype modifications from allowing a guest to breach containment.

5 Blancura

Current statically verified JavaScript subsets are not robust to prototype extensions. If a web site extends a built-in prototype object with a new method, guest code can access that method and, if that method is exploitable, breach the subset's containment. We propose further restricting guest code to prevent this kind of containment breach. Instead of using a blacklist to ban known-dangerous properties, our

```
String.prototype.extractScripts = function() {
  var matchAll = new RegExp(
      Prototype.ScriptFragment, 'img');
  var matchOne = new RegExp(
      Prototype.ScriptFragment, 'im');
  return (this.match(matchAll) || []).
      map(function(scriptTag) {
        return (scriptTag.match(matchOne) ||
                ['', ''])[1];
      });
}
String.prototype.evalScripts = function() {
  return this.extractScripts().map(
      function(script) { return eval(script) });
}
```

(a) Relevant host code

```
<div id="GUESTAD_">
<script>
  "use strict";
  ADSAFE.go("GUESTAD_", function (dom, lib) {
    var expl = '<script language="javascript">' +
        '... attack code ...<\/script>';
    expl.evalScripts();
  });
</script>
</div>
```

(b) ADsafe-verified guest code implementing exploit

Figure 5. Exploit for `twitter.com`

system, Blancura, whitelists known-safe properties. In our system, the guest cannot access exploitable methods defined by the host because those functions are not on the whitelist.

5.1 Design

To improve isolation, we propose running the host and each guest in separate JavaScript namespaces. By separating the namespaces used by different parties, we can let the host and guest interact with the same objects (e.g., the string prototype object) without interfering with one another. Essentially, our system whitelists access to property names within the proper namespace and blocks access to all other property names. This approach is similar to script accenting [11] and the "untrusted" HTML attribute from [18], both of which separate the namespaces for each distinct principal. For example, we prohibit the guest advertisement from containing the following code:

```
obj.foo = bar;
```

Instead, we require `foo` have a page-unique prefix:

```
obj.BLANCURA_GUEST1_foo = bar;
```

The Blancura verifier enforces this property statically by parsing the guest's JavaScript and requiring that all property names begin with the prefix. For dynamic property accesses (i.e., with the `[]` operator), Blancura enforces namespace separation dynamically using the same mechanism ADsafe uses to blacklist property names.

To generate the page-unique prefix, we leverage the fact that ADsafe guests already contain a page-unique identifier: the widget ID. ADsafe requires that guest JavaScript code appears inside a `<div>` tag with a page-unique `id` attribute, which ADsafe uses to identify the widget. We simply use this value, prefixed by `BLANCURA_`, as the guest's page-unique namespace. This technique lets us avoid using randomization to guarantee uniqueness, dodging the thorny problem of verifying statically that a piece of code was "correctly randomized."

Using separate namespaces protects hosts that add vulnerable methods to built-in prototype objects. For example, if the host page adds a vulnerable `right` method to the `String` prototype, a malicious guest would be unable to exploit the method because the guest is unable to access the `right` property. In some sense, this design is analogous to doing an access check on every property access where the property name prefix is the principal making the access request. Ideally, we would modify the host to use a separate namespace (e.g., `BLANCURA_HOST_`), but advertising networks are unable to rewrite the publisher's JavaScript. In practice, however, we do not expect publishers to add properties to the built-in prototypes with guest prefixes, an expectation which is validated by our experiment.

After placing the guest in its own namespace, the guest is unable to access any of the built-in utility methods because those methods have not yet been whitelisted. However, many of those functions are useful and safe to expose to guest script. To expose these methods to the guest, the Blancura runtime adds the appropriate property name:

```
String.prototype.
  BLANCURA_GUEST1_indexOf =
    String.prototype.indexOf;
```

The guest code can then call this function as usual using its prefixed property name, incurring negligible runtime and memory overhead. Instead of blacklisting dangerous built-in methods such as `concat`, this approach lets the subset designer expose carefully vetted methods to the guest individually instead of punting the issue to the developers who use the subset.

Adding prefixes to property names does not measurably affect runtime performance. When the JavaScript compiler parsers a JavaScript program, it transforms property name

strings into symbolic values (typically a 32-bit integer), and the symbols generated with and without the prefix are the same. Whitelisting existing properties of a built-in object does incur a tiny memory overhead because more entries are added to the object's symbol table, but this memory overhead amounts to only a handful of bytes per property. Emulating DOM interfaces that use getters and setters (such as `innerHTML`) incurs some run-time cost because the runtime must install a custom getter or setter with the prefixed name. However, systems like ADsafe often wrap the native DOM with a library free of getters and setters [14].

5.2 Implementation

We implemented Blancura by modifying the ADsafe verifier. Using a 43 line patch, we replaced the blacklist used by ADsafe's static verifier with a whitelist. By making minimal modifications to the ADsafe verifier, we have a high level of assurance that our implementation is at least as correct as ADsafe's implementation, which has been vetted by experts in a public review process. Additionally, Blancura is a strict language subset of ADsafe, ensuring that any vulnerabilities in Blancura are also vulnerabilities in ADsafe. Our verifier is available publicly at `http://webblaze.cs.berkeley.edu/2010/blancura/`.

To demonstrate that Blancura is as expressive as ADsafe, we implemented a source-to-source compiler that translates ADsafe widgets into Blancura widgets by prefixing each property name with the appropriate namespace identifier. We implemented our compiler by modifying the Blancura verifier to output its parsed representation of the widget. Our compiler is idempotent: if the input program is already in the Blancura subset, the output of the compiler will be identical to the input. The compiler can, therefore, also be used as a Blancura verifier by checking whether the output is identical to the input.

Although we have based our implementation of Blancura on ADsafe, we can apply the same approach to the other JavaScript subsets that use static verification, such as Dojo Secure or Jacaranda. In addition, our approach of whitelisting property names is also applicable to FBJS even though FBJS is based on a source-to-source compiler and not a static verifier. In each case, using the Blancura approach improves the security of the system with minimal cost.

5.3 Imperfect Containment

Despite improving the containment offered by static verifiers, a host page can still compromise its security and let a guest advertisement breach containment because JavaScript programs can call some methods implicitly. For example, the `toString` and `valueOf` methods are often called without their names appearing explicitly in the text of the program. If the host page overrides these methods with vulnerable functions, a malicious guest might be able to breach containment. In our experiment, the only site that overrode these methods was Facebook, which replaced the `toString` method of functions with a custom, but non-exploitable, method.

A host page can also compromise its security if it stores a property with a sensitive *name* in a built-in prototype object because guests can observe all the property names using the `for(p in obj)` language construct. Similarly, a guest can detect the presence of other guests (and their widget identifiers) using this technique. We could close this information leak by banning this language construct from Blancura, but we retain it to keep compatibility with ADsafe and because, in our experiment, we did not observe any sites storing sensitive property names in prototype objects.

6 Conclusions

Existing secure JavaScript subsets that use statically verified containment blacklist known-dangerous properties. This design works well on the empty page where the set of functions accessible from the built-in prototype objects is known to the subset designer. However, when verified advertisements are incorporated into publisher web sites, the advertisement can see methods added to the built-in prototypes by the publisher. If the publisher does not carefully vet all such functions, a malicious advertisement can use these added capabilities to breach the subset's containment and compromise the publisher's page.

To determine whether this infelicity is a problem in practice, we analyzed the non-adult web sites from the Alexa Top 100 most-visited web sites in the United States. We found that over a third of these web sites contain functions that would be exploitable by an ADsafe-verified advertisement displayed on their site. In fact, we found that once a web site adds a non-trivial number of functions to the built-in prototypes, the site is likely to be exploitable. From these observations, we conclude that JavaScript subsets that use statically verified containment require improvements before they can be deployed widely.

We propose improving these systems by revising a design choice. Instead of blacklisting known-dangerous properties, we suggest whitelisting known-safe properties by restricting the guest advertisement to accessing property names with a unique prefix. This mechanism prevents advertisements from accessing properties installed by the host page while maintaining the performance and deployment advantages of static verification. With our proposal, the web sites in our study do not need to be modified in order to host untrusted advertisements securely.

Acknowledgements. We would like to thank Douglas Crockford, David-Sarah Hopwood, Collin Jackson, Mark Miller, Vern Paxson, Koushik Sen, Dawn Song, David Wagner, and Kris Zyp for feedback and helpful conversations throughout our work on this project. This material is based upon work partially supported by the National Science Foundation under Grant No. 0430585 and the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170.

References

[1] Alexa - Top Sites in United States. `http://alexa.com/topsites/countries/US`.

[2] google-caja. `http://code.google.com/p/google-caja/`.

[3] Live Labs Web Sandbox. `http://websandbox.livelabs.com/`.

[4] Who's using Prototype. `http://www.prototypejs.org/real-world`.

[5] Caja Test Bed, August 2009. `http://cajadores.com/demos/testbed/`.

[6] HTML 5: A vocabulary and associated APIs for HTML and XHTML, April 2009. `http://www.w3.org/TR/2009/WD-html5-20090423/`.

[7] Performance of cajoled code, August 2009. `http://code.google.com/p/google-caja/wiki/Performance`.

[8] Prototype JavaScript Framework, May 2009. `http://www.prototypejs.org/`.

[9] Adam Barth, Collin Jackson, and William Li. Attacks on JavaScript Mashup Communication. In *Web 2.0 Security and Privacy Workshop 2009 (W2SP 2009)*, May 2009.

[10] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proc. of the 18th USENIX Security Symposium (USENIX Security 2009)*. USENIX Association, August 2009.

[11] 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.

[12] Steven Crites, Francis Hsu, and Hao Chen. OMash: enabling secure web mashups via object abstractions. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 99–108, New York, NY, USA. ACM.

[13] Douglas Crockford. ADsafe. `http://www.adsafe.org`.

[14] Douglas Crockford. ADsafe DOM API. `http://www.adsafe.org/dom.html`.

[15] David-Sarah Hopwood. Jacaranda Language Specification, draft 0.3. `http://www.jacaranda.org/jacaranda-spec-0.3.txt`.

[16] David-Sarah Hopwood. Personal Communication, June 2009.

[17] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: secure component model for cross-domain mashups on unmodified browsers. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 535–544, New York, NY, USA. ACM.

[18] Adrienne Felt, Pieter Hooimeijer, David Evans, and Westley Weimer. Talking to strangers without taking their candy: isolating proxied content. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 25–30, New York, NY, USA, 2008. ACM.

[19] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proc. of the 18th USENIX Security Symposium (USENIX Security 2009)*. USENIX Association, August 2009.

[20] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. MashupOS: operating system abstractions for client mashups. In *HotOS '07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–7, Berkeley, CA, USA. USENIX Association.

[21] Collin Jackson and Helen J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 611–620, New York, NY, USA, 2007. ACM.

[22] Dan Kaplan. Malicious banner ads hit major websites, September 2007. `http://www.scmagazineus.com/Malicious-banner-ads-hit-major-websites/article/35605/`.

[23] S. Maffeis, J.C. Mitchell, and A. Taly. Run-time enforcement of secure javascript subsets. In *Proc of W2SP'09*. IEEE, 2009.

[24] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[25] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] Microsoft Live Labs. Microsoft Web Sandbox. `http://www.websandbox-code.org/Samples/genericSample.aspx`.

[27] Elinor Mills. Malicious Flash ads attack, spread via clipboard, August 2008. `http://news.cnet.com/8301-1009_3-10021715-83.html`.

[28] Ashlee Vance. Times Web Ads Show Security Breach, September 2009. `http://www.nytimes.com/2009/09/15/technology/internet/15adco.html`.

[29] Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the web. In *WWW '09: Proceedings of the 18th international conference on World Wide Web*, pages 961–970, New York, NY, USA, 2009. ACM.

[30] Kris Zyp. Secure Mashups with dojox.secure. `http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/`.

A Alexa US Top 100

We list here the subset of the sites in the Alexa US Top 100 sites that we used in our experiment. Note that the subset does not include the nine adult sites listed at the time of access, nor does it include `blogger.com` because it is the same site as `blogspot.com`. Alexa was accessed on May 1, 2009 to obtain the list of sites.

Rank	Website	# Suspicious Edges
1	google.com	0
2	yahoo.com	0
3	facebook.com	17
4	youtube.com	2
5	myspace.com	36
6	msn.com	14
7	live.com	0
8	wikipedia.org	0
9	craigslist.org	0
10	ebay.com	13
11	aol.com	9
12	blogspot.com	13
13	amazon.com	2
14	go.com	0
15	cnn.com	33
16	twitter.com	60
17	microsoft.com	35
18	flickr.com	12
19	espn.go.com	1
20	photobucket.com	60
21	wordpress.com	0
22	comcast.net	62
23	weather.com	0
24	imdb.com	0
25	nytimes.com	61
26	about.com	0
27	doubleclick.com	1
29	linkedin.com	0
30	apple.com	16
31	cnet.com	72
32	verizon.net	0
33	vmn.net	18
34	netflix.com	17
35	hulu.com	63
36	mapquest.com	5
37	att.net	0
38	rr.com	70
39	adobe.com	59
40	foxnews.com	52
42	ask.com	0
43	mlb.com	3
44	rapidshare.com	0
45	answers.com	3
46	walmart.com	0
48	fastclick.com	0

Rank	Website	# Suspicious Edges
51	reference.com	10
52	bbc.co.uk	0
53	target.com	0
54	tagged.com	44
55	ning.com	0
56	careerbuilder.com	37
57	dell.com	0
59	disney.go.com	0
60	digg.com	1
61	att.com	0
62	usps.com	63
63	typepad.com	0
64	wsj.com	64
65	ezinearticles.com	0
66	bestbuy.com	1
67	foxsports.com	0
68	livejournal.com	20
69	thepiratebay.org	14
70	ehow.com	13
71	imageshack.us	2
72	tribalfusion.com	0
74	aweber.com	0
75	ups.com	0
76	megavideo.com	0
77	yelp.com	66
78	mozilla.com	0
79	deviantart.com	9
80	expedia.com	0
82	pandora.com	0
83	nba.com	61
84	newegg.com	28
86	irs.gov	0
87	washingtonpost.com	0
88	cox.net	0
89	dailymotion.com	72
91	download.com	72
92	reuters.com	2
93	zedo.com	2
94	monster.com	17
95	people.com	1
96	verizonwireless.com	37
97	realtor.com	1
98	ign.com	9
99	pogo.com	0
100	latimes.com	1

Table 2. Subset of the Alexa US Top 100 sites used in the experiment.