Protecting Private Web Content from Embedded Scripts

Yuchen Zhou and David Evans

University of Virginia
{yuchen,evans}@virginia.edu

Abstract. Many web pages display personal information provided by users. The goal of this work is to protect that content from untrusted scripts that are embedded in host pages. We present a browser modification that provides fine-grained control over what parts of a document are visible to different scripts, and executes untrusted scripts in isolated environments where private information is not accessible. To ease deployment, we present a method for automatically inferring what nodes in a web page contain private content. This paper describes how we modify the Chromium browser to enforce newly defined security policies, presents our automatic policy generation method, and reports on experiments inferring and enforcing privacy policies for a variety of web applications.

1 Introduction

Web applications can provide better services and more targeted information by customizing content for individual users. Those customizations, however, may leak personal information to third parties whose scripts are embedded in the web page. Current web browsers grant embedded scripts full access to all content on the page, including the ability to access any personal profile information, photos, email addresses, or other private content that is displayed in the web page. Many commonly used scripts require host pages to directly embed their scripts into the host page. Scripts that must be directly embedded include popular ad networks (including Google AdSense and Yahoo! Advertising), analytics scripts (including Google Analytics), and Facebook's recently released comments API [18]. One example of how this privilege could be abused is an advertisement embedded in Facebook pages last year that offended privacy expectations by incorporating images of the user's friends in an advertisement [17].

The easiest way to isolate untrusted scripts from the host page is to put them in an iframe. Since a script included using an iframe comes from a different origin, that script cannot access any resource in the host domain. This isolation is complete—the included script cannot interact with any other part of the page. To avoid the all or nothing model, several researchers have proposed alternatives that provide untrusted scripts with limited access to the host. As we discuss further in Section 6, though, none of these solutions satisfactorily address the security, functionality, and usability requirements necessary for a solution to be widely deployed.

Threat Model. We focus on the scenario where a content provider wants to embed content from untrusted third-party scripts such as advertising, analytics scripts, and gadgets in its output pages that contain private user information. The adversary controls one or

V. Atluri and C. Diaz (Eds.): ESORICS 2011, LNCS 6879, pp. 60–79, 2011.

more of the scripts embedded in the target page. To obtain private content, the embedded script may use any means provided by JavaScript to get the text or attribute of a confidential node including directly calling DOM APIs or probing values of variables in host scripts. We assume a one-way trust model since our goal is to protect user content from untrusted scripts rather than to protect embedded scripts from the host page or each other. Host scripts should be able to access the full functionality of third-party scripts, but third-party scripts should not be able to access or modify host scripts. Hence, we provide a form of one-way access from host to guest scripts. In summary, our goal is to provide third-party scripts with limited access to the DOM and no access to host scripts, while granting host scripts full access to third-party scripts and the DOM.

We do not target JavaScript frameworks such as jQuery that require rich, bi-directional interactions with the host's content. In these cases, we assume the developers fully trust the third-party libraries. We also do not consider other attack vectors such as cross-site scripting attacks or web browser vulnerabilities. Many other projects have focused on mitigating these risks, and we concentrate on the scenario where the host page developer deliberately embeds untrusted scripts.

Contributions. Our approach has three main advantages over previous approaches:

Fine-grained access control. Polices can be specified at a per-node, per-script granularity. For example, we allow the host page to set DOM node A to be invisible to script X, while DOM node B is read-only to script Y and fully accessible to script X. Developers can explicitly allow scripts to collaborate and execute in the same context while isolating them from other scripts on the page. This provides greater flexibility and expressiveness than previous solutions. For example, MashupOS [22] and Jayaraman et al. [10] base their policies on node locations on the page. We also provide a mechanism that gives developers one-way access to untrusted JavaScript code without exposing trusted scripts. Section 2 explains the policies enabled by our mechanisms.

Compatibility. Previous approaches place restrictions on what embedded scripts may contain, often placing limits on dynamic script execution. For example, AdJail [12] and Stamm et al. [19] do not support script node insertion; Caja [14] and AdSafe [3] do not support `eval`. AdJail [12] also does not fully support `document.write()`. We avoid JavaScript source code transformations to ensure maximum compatibility and performance. Our approach allows embedded scripts to use all of JavaScript with no restrictions, except those imposed by the actual access control policy. Section 3 explains how our implementation achieves this.

Easy deployment. One of our goals is to enable developers to painlessly incorporate our protection into legacy web applications, so our approach minimizes the effort required from the developer. All developers need to do is identify untrusted scripts (which can usually be done automatically based on their origin) and annotate nodes that contain private information by adding an attribute to that node. To further reduce deployment effort, we developed a method for automatically identifying all the nodes in a web page that may contain private information (Section 4). For this, we consider any content that varies depending on whether the page is requested with or without the user's credentials as private. We envision automatic policy learning as part of a third-party or ISP service, enabling our protections to be provided without any cooperation from sites.

We evaluate our design by implementing it as a modification to the Chromium browser, and conducting both security and functionality experiments on a range of websites. Section 5 reports on our experiments that show it is possible to automatically learn effective privacy policies for most tested sites, and to enforce our isolation and privacy mechanisms without requiring developer modifications or breaking website functionality.

2 Protecting Private Data

We provide two types of protection policies: JavaScript execution isolation and DOM access control.

2.1 Execution Isolation

One of our primary goals is to let web developers easily group third-party scripts so that some of them may collaborate with each other while still remaining separated from other third-party scripts and host page scripts. To facilitate this we add a new attribute to the script tag: worldID=*string*. This idea originates from Barth et al.'s *isolated world* concept [1] which was developed to isolate browser extensions. Each world with a unique worldID is isolated from all other worlds. The worldID attribute also serves as the principal for scripts for controlling access to DOM nodes (Section 2.2).

Figure 1 illustrates the semantics of the worldID attribute. The custom and native objects of the first script (in worldID="1") are isolated from the second script because they have different worldIDs. This means the variable a, defined in the first script, is not visible in the second script, and the second script only sees the original toString method.

```
<script worldID = "1">
  var a = 3;
  function f() {}
  Boolean.prototype.toString = f;
</script>
<script worldID = "2">
  var b = a; // error: a undefined
  f(); // error: f undefined
  new Boolean(0).toString();
      // calls original toString
</script>
<script worldID = "1">
  var b = a; // OK
  new Boolean(0).toString(); // f()
</script>
```

Fig. 1. Execution context separation

```
<div id="a" RACL="1,2" WACL="1">
  User: Alice
</div>
<script worldID = "1">
  var b=document.getElementById('a'); // OK
  b.innerHTML = 'changed'; // OK
</script>
<script worldID = "2">
  var b=document.getElementById('a'); // OK
  b.innerHTML = 'changed'; // disallowed write
</script>
<script worldID = "3">
  var b=document.getElementById('a');
      // error: a not readable
</script>
```

Fig. 2. DOM access mediation

Since the third script has worldID="1", it executes within the same context as the first script and can access all the objects the first script can.

Shared Libraries. Full isolation of embedded scripts would break the functionality of many host pages. To support embedded scripts that are used as libraries, we added two new attributes to script tags: sharedLibId and useLibId. All objects inside a script tagged with a sharedLibId attribute can be accessed by the host execution context as well as all other worlds that have the corresponding useLibId attribute. The third-party scripts, however, cannot access the privileged scripts and are still bound by the DOM access policies.

For example, *Google Analytics* users can use _gaq to track business transactions. The host script pushes transaction information into the array _gaq which is later processed by Google Analytics. Now that we have isolated the context, the _gaq variable would not normally be visible in other worlds. To support this, the sharedLibId attribute is defined to identify when an embedded script is a shared library:

```
<script src="google.com/GA.js" worldID="1" SharedLibId="GA">
```

Then, other scripts can use the useLibId attribute to access objects defined in the shared library. To prevent pollution of other script objects, objects in the shared library are prefixed with the library identifier. For example,

```
<script useLibId = "GA">
    GA._gaq.push(['_addTrans', '1234', '11.99']);
</script>
```

2.2 DOM Node Access Control

In addition to isolating objects in scripts, we provide fine-grained access control over host objects at the granularity of DOM nodes. We introduce two additional tags for all nodes in the DOM tree: RACL for specifying read access, and WACL for specifying write access. Each access control list is a comma-separated list of worldIDs. Only scripts running in the worlds listed in the RACL list are permitted to read the node, and only scripts listed in WACL are permitted to modify the node. For example, if a third-party script wants to remove a node, it must have the privileges of modifying both that node and its parent (this is consistent with the JavaScript syntax for removing a node which requires two node handles: parentNode.removeChild(thisNode)). On the other hand, to append a node to an existing node, a script only needs to have write privileges for the parent node since it already has access to the node to be inserted. The ACLs a node has do not depend on its parent or children.

As shown in Figure 2, a script can only access a particular div element if it is present in the corresponding access control list of that element. This is more flexible than previous works like Adjail [12] and MashupOS [22]. Table 1 summarizes the customizable policies for providing fine-grained mediation of host objects together with the control of sharing and isolation of custom and native objects.

Special Root Properties. In addition to specific DOM nodes, we also provide a way to hide selected APIs from certain scripts. These special host objects may provide scripts

Table 1. Summary of Policy Attributes

Context	Policy syntax	Semantics
script	worldID="s"	WorldID of the script context is s
script	sharedLibId="s"	This is script from s library
script	useLibId="s"	This script requires to use s library
DOM node	RACL="$d_1, d_2, \ldots$"	Worlds that may access this
DOM node	WACL="$d_1, d_2, \ldots$"	Worlds that may modify this

with access to private information. For example, document.cookie returns authentication tokens. Since cookie is a special property of the document it is not associated with any specific node. Other examples include document.location, document.URL and document.title as well as powerful APIs such as document.write() and document.open(). Therefore, we add a set of new attributes for the <html> tag to allow developers to specify these per-API/per-script policies. These privileges are disallowed for untrusted scripts unless explicitly permitted.

3 Implementation

Our implementation is built on Google's open source Chromium project (revision 57642 on Windows 7). Approximately 1500 lines of code were added or modified, mostly in the WebKit DOM implementation and the bindings of V8 JavaScript interpreter and WebKit DOM. We did not modify V8. Hence, our implementation could be adapted to other browsers that use WebKit as well with the effort of adding *isolated world* support.

Figure 3 illustrates how a DOM API call is executed in our system. In step 1, the WebKit parser parses a raw HTML file from a remote server and passes each script node to the *ScriptController* in WebKit/V8 bindings to set up the execution environment. If the context associated with the current worldID is already created, *ScriptController* tells V8 to enter, otherwise it creates a new one. In step 2, the *ScriptController* sends the script to V8 to start script execution. At some point, V8 encounters a DOM API call and invokes a callback to the corresponding function using the WebKit/V8 bindings (step 3).

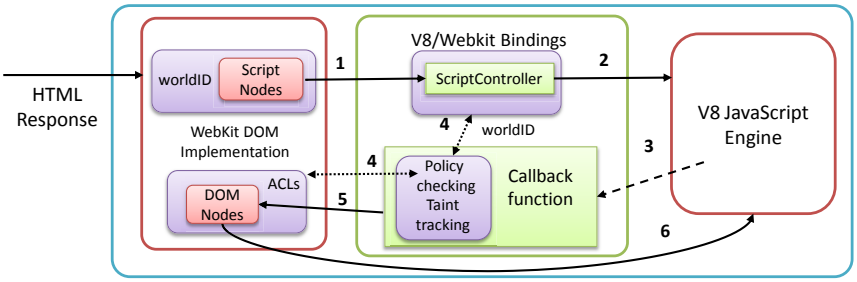

Fig. 3. Execution flow of a DOM API call

In step 4, that callback function is modified to include policy checking code that checks the worldID against the ACLs of the node. After passing the policy checking, the call is forwarded to the WebKit DOM implementation (step 5). In cases where modification happens, the target node is also tainted according to rules explained in Section 3.3. Finally, the result is returned from the WebKit DOM back to V8 (step 6). Next, we provide details on how we enforce script isolation and mediate access to the DOM. Section 3.4 discusses some special issues for handling dynamically-generated scripts.

3.1 Script Execution Isolation

Isolating any two scripts by putting them into different execution contexts allows us to specify per-script policies. We adopt Barth et al.'s *isolated world* mechanism [1]. This is used in Chrome to separate the execution context of different browser extensions, so a security compromise of one extension does not compromise the host page or other extensions. The isolated world mechanism replaces the one-to-one DOM-to-JS execution context mapping with a one-to-many map where each context maintains a mapping table to the DOM elements of the host page. This ensures that only host objects are shared among all worlds, but not native or custom objects. If a script in world 1 declares a local variable or modifies the toString prototype function, it is not visible to other worlds. If that script changes host page DOM elements, though, the changes are propagated to all other worlds (our policy mechanisms can disallow such modifications).

We extend this mechanism to apply to embedded scripts instead of just extension content scripts. We modified Chrome to recognize a new attribute worldID so that the WebCore ScriptController can support different JavaScript execution contexts according to a scripts' worldID. A hashmap of all the execution contexts is instantiated on a per-page basis to enable scripts to execute in the correct context. Two scripts with different worldIDs run in completely different contexts and cannot access each other's objects.

Host Script Access. For compatibility, we also need an asymmetric way for host scripts to access third-party objects. We take advantage of two properties: (1) all objects defined in the script are children of a global object, DOMWindow; and (2) it is possible to inject arbitrary objects into another context using Google V8 JavaScript engine APIs. We modified the browser to automatically grab the handle of the global object of that context and inject it into the host context as soon as a third-party script execution context with a SharedLibId is created. As long as the global objects of trusted contexts are never passed to untrusted contexts, third-party scripts are never able to access objects in trusted scripts. Here, developers need to be careful not to pass any confidential host objects to untrusted scripts.

3.2 DOM Access Control

Fine-grained policies allow different scripts to be granted different access permissions. We do this by either hiding inaccessible nodes from scripts based on their worldID, or in cases where more expressive policies are needed, by mediating access requests.

Completely Hidden Nodes. Unreadable nodes can be completely hidden from scripts. In our implementation, attempts to request a reference of a hidden node or any API on a hidden node instead receive a fabricated result. We return the v8::null() object for

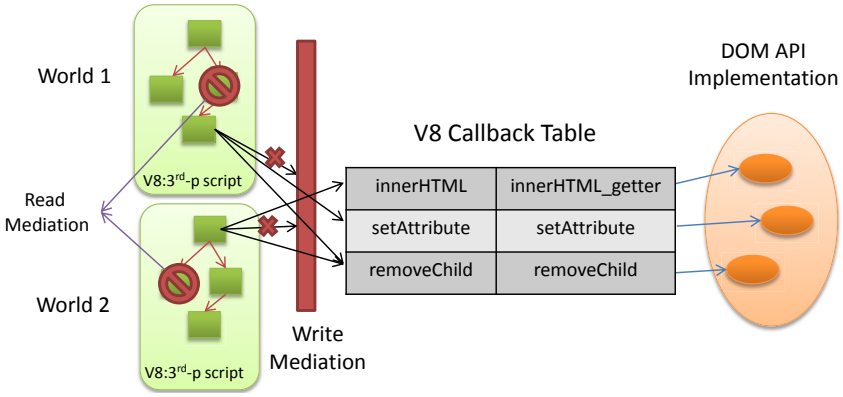

Fig. 4. JavaScript to DOM API Mediation

functions that would normally return a DOM node wrapper; we return an empty string object for functions that would normally return a string object. The null results avoid leaking any information, but should enable a well-written script to continue (we confirm this in our experiments, as reported in Section 5.2).

Our implementation mediates all DOM API getter functions to check the ACL of target node as shown in Figure 4. The upperleft and the lowerleft squares indicate two different execution worlds. As each world tries to grab handles of different nodes or call getterAPIs on those nodes, some of them are thwarted by our mediation according to respective policies; the ones that get through are executed normally.

Mediated functions include all the node handler getters as well as APIs that can be called after a node handler is held, such as getAttribute(). One of the trickier APIs to deal with is the innerHTML getter, as well as similar APIs. These APIs are designed to return the text/HTML markup of all children of this node. AdJail[12] does not have to worry about this since their policies require that the parent of a subtree cannot be assigned more privileges than the intersection of its children's privileges. Since in our case some of the children may have been marked private while the root node is marked public, calling the innerHTML getter on parent nodes may reveal confidential information in its children. To remedy this, we modify the implementation of the innerHTML callback and other similar APIs to filter out private nodes from the result.

Read-Only Access. Providing read-only or other restricted access is more complex since it requires giving the script a handle to the node. There are five ways a script may modify a node: 1) directly changing a node property (Chrome calls the internal setter function), 2) modifying the style of that node, 3) modifying the children of that node, 4) modifying the attribute of that node by calling node-specific JS-DOM APIs (e.g., setAttribute(), textContent), or 5) attaching or removing any event handlers to that node (e.g. addeventhandler()). Each of these is handled in a completely different fashion in Chromium, so it is necessary to address all of them.

We modified all related JS-DOM binding functions and made sure that if a script's worldID does not appear in the WACL of a node it cannot do any of these actions. Special

caution has to be used when coping with textNode because the browser exposes a quite different set of APIs. The security attributes, WACL, RACL, and worldID should never be changed by scripts other than the host since this would allow untrusted scripts to change the policy. We therefore modified the attribute setters to check attribute names and the script's worldID to prevent unauthorized modifications to these attributes.

3.3 Taint-Tracking

Since a node may initially contain public information, but later be modified by a script to contain private information. This use of Ajax/XHR to dynamically authenticate users and update respective content is not uncommon among the sites we have tested (for example, `cnn.com` uses JavaScript to update the username box on the upper right corner of the page after the entire page is loaded). Thus, it is important to update the privacy status of a node when it is modified by a script. We do this using a conservative taint-tracking technique that marks a DOM node as private whenever any host script modifies it. Nodes that are modified by a script with worldID=*a* are only visible to scripts in world *a* as well as the host scripts.

We implemented a simple taint-tracking design that automatically marks a node as private when it is changed by a host script. Since our experiments show that this tainting policy occasionally leads to compatibility problems when too many nodes are tainted, we relaxed tainting by adding a heuristic to only taint nodes whose text content or source attributes are changed by the script. This lowers the false positive rate by ignoring the CSS and location changes of the nodes. In case this policy is too relaxed for certain websites, developers can manually mark these nodes as private using the WACL or RACL attributes. This heuristic does not pose a privacy risk, but enables side-channels between scripts that could otherwise not communicate since they may be able to modify a node that can be read by the other script. We do not consider this a serious security risk since private data is only exposed to a third-party when explicitly allowed by the policy, so although that script can now leak the data to a different third-party script it could also misuse the data directly.

3.4 Dynamic Scripting

Many previous works feared the consequences of allowing dynamically-generated code and simply excluded dynamic parts of JavaScript such as eval. This fear is justified for any rewriting-based approach since dynamically-generated code circumvents the rewriting protections. Since we enforce policies at run-time, we can fully support dynamic scripts but need to be careful to assign the appropriate policies to generated scripts. In particular, generated scripts may execute in different contexts from the scripts that created them. This may break functionality since variables and functions that should be shared are now isolated. More seriously, it may also lead to privilege escalations if less privileged scripts are able to dynamically create a higher privileged script.

We solve both problems by propagating worldIDs. Dynamically-generated scripts inherit the worldID from their creator, thus executing within the same context. We mediate all four ways to dynamically evaluate a script: 1) calling eval() or setTimeOut(), etc.; 2) defining an anchor element with JavaScript pseudo-protocol (i.e., javascript:code;); 3) creating a script node with arbitrary code; or 4) embedding a new script node by calling

document.write(). The first two cases are handled by modifying respective script initialization functions in the V8ScheduledAction and ScriptController class. For the third case, we strip any worldID attributes from created node and add the creator's worldID attribute. This is done automatically inside the browser. The fourth situation is most complex, and discussed next.

Injected Scripts. In the fourth situation, scripts may be added to the page dynamically using document.write or document.writeln. These functions can dynamically create scripts by injecting raw HTML code into the page. These interfaces are very powerful, but it is necessary to support them to maintain compatibility with many existing web applications. To address this, we inject several lines of code in the HTML parser to ensure the parser correctly interprets the current execution context and then adds the appropriate worldID attribute to dynamically-created script nodes.

Event Handlers. Third-party scripts may also insert code in the context of host scripts by adding that code as an event handler of another DOM node, assuming the event can be triggered (e.g., using the onload event). There are four possible ways to attach an event handler: 1) direct assignment (e.g., someNode.onclick = 'somefunction()'), 2) setAttribute, 3) addEventListener, or 4) creating an attribute node and attaching it to a node (e.g.,). To preserve policy enforcement and execution context, an event handler should execute in the same context as the script that created it. For each of the four ways of attaching event handlers, we propagate the worldID to make sure that the event handler executes in the correct context. Note that after the host script registers an event handler, third-party scripts can try to call that event handler even if the event is not triggered. Hence, we associate all event handlers with their creator's worldID and mediate all the getters of event handlers to make sure the caller's worldID is identical to the callee's.

4 Automatic Policy Generation

To protect private information in host pages, we need some way to identify what nodes in the host page contain private information. This could be done by web application developers manually annotating nodes as public or private. Manual annotation, however, is probably too tedious for most web applications and unlikely to happen until a protection system is widely deployed. If we had access to the server, one strategy would be to use information flow techniques at the server to track private content and mark nodes containing private content when they are output. Since we do not assume server access, however, here we instead present a dynamic technique for inferring private content solely based on the pages returned from different requests.

We define *private content* as any content that varies depending on user credentials. Thus, any content that is different in an authenticated session from what would be retrieved for the same request in an unauthenticated session is deemed private. Nodes that directly contain private information should be marked private, but not the parent of that node. For example, if *Username* appears, only the inner span element is private, but not the outer div. The fine-grained nature of our policy enforcement supports this. We automate learning policies by submitting multiple requests

to the server with different credentials, and identifying the differences as potentially private content.

One of our design goals is to minimize the changes have to be made both on server side and on client side, so we use a proxy to add security policies. Figure 5 illustrates the structure of our policy learner. The proxy automatically identifies third-party scripts and generates the policies for the response when a request is captured. The resulting page, including the inferred policies, is passed on to browser client.

Our proxy is implemented using Squid, which supports the Internet Content Adaptation Protocol (ICAP) that allows us to modify web traffic on the fly. For convenience, we run the Squid server in the same machine as our modified browser, however it could be moved onto an intermediate node along the routing path for better centralized control. For the ICAP server implementation, we use GreasySpoon [15]. This design cannot deal with SSL web traffic since the proxy will only see the encrypted traffic. The Chromium development group is currently (as of June 2011) still working to implement webRequest and webNavigation as experimental extension APIs [6]. Once these are implemented, we can move our proxy server inside the browser thus making it work on SSL/TLS traffic and easing deployment.

The content adaptation is divided into two main functions: third-party script identification and public node marking.

Third-party Script Identification. The ICAP server examines the response header. For each script with a source tag we compare the script's source with the host domain. For scripts that come from different domains, we add worldID attributes that identify the origin and indicate that they are not trusted by the host.

Identifying Public Nodes. To identify public content, our proxy compares the responses from two requests, one with the user's credentials and one without, and denotes any content that is identical in both responses as public. For example, assume a user visits `nytimes.com` so the browser sends a request including the user's cookies as credentials to `nytimes.com` and stores this response as R_{priv}. Once our ICAP server sees the incoming response it sends the same request except without including the cookies, storing the response as R_{pub}.

Once it has both responses, the proxy executes a differencing algorithm. This is similar to a simple text diff, except it follows the node structure. Initially, all nodes

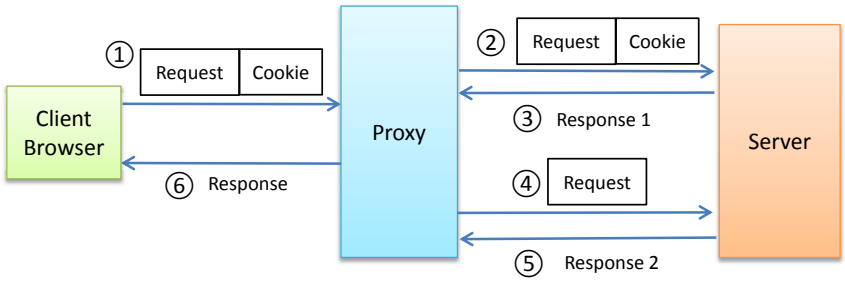

Fig. 5. Automatic Policy Generation

are assumed to be private. Then, any node in R_{priv} that appears identically in R_{pub} is marked as public. For write accesses, we make sure all children of a root node are the same before marking the root node public. Read access is slightly more relaxed than write access, since we already modified innerHTML function to conceal private nodes inside a subtree. As long as the attributes and immediate textnode children are the same in both responses we mark that node public.

State-Changing Requests. Our policy learning process requires sending duplicate requests to the server. This could have undesirable side effects if an unauthenticated request can alter server state. To limit this, POST requests are ignored since sending them twice could result in undesired state changes at the server. The entire response from a POST request is considered private. The HTML specification suggests GET methods should be idempotent [4], but many sites do make persistent state changes in response to GET requests. For example, a forum site might use a GET request for anonymous postings. If we submit the request twice the anonymous comment may be posted twice since no credentials are required for the posting.

We consider two possible solutions. The first is for the server to annotate non-idempotent pages. The first time a user visits a site the proxy has not seen before, it skips the request duplication and looks for idempotent field in the response header. Servers can send idempotent=**false** in the header to indicate that the browser should not to send duplicate requests for this page. If the idempotent field is not detected in the first response we resume the proxy behavior and submit the duplicate requests.

A second approach is to set up a third-party service like AdBlock and have users subscribe to this service. The centralized server collects information from users and correlates responses to mark private data. If we have an authority like this we do not necessarily need to send two requests since other users may have already submitted similar requests and the server should already have recorded the responses. This centralized server should be established at the ISP so that we do not introduce extra vulnerable point in the network path. Of course in this case the ISP server's identification accuracy would affect many more users than a local proxy, but it is also convenient to manually correct the mistakes as a center server. This approach also has a drawback that the requests may not necessarily happen near each other in terms of timing. For a highly active news site like `nytimes.com`, the structure or content might change fast enough that more false positives will appear.

5 Evaluation

We evaluated security or our implementation by manually testing a range of possible attacks, its compatibility with a sample of web applications, and the effectiveness of the automatic policy generator.

5.1 Security

We tested our implementation against all attack vectors we could identify from the W3C DOM [21] and ECMA specifications [9]. Table 2 lists the attack vectors and examples of the attacks we tested. For each attack vector, we created at least one test case for

Table 2. Attack Space Summary

Attack Type	Examples
Calling DOM to get nodes	document.getElementById(), nextSibling(), window.nodeID
Calling DOM to modify nodes	nodeHandler.setAttribute(), innerHTML=, nodeHandler.removeChild()
Probing host for private objects	reading host vars, calling host functions and event handlers
Accessing special properties	document.cookie, open(), document.location

each feature in the W3C DOM/ECMAscript specification and confirmed that the attack is thwarted by our implementation. Since most of these attack vectors are handled by a few functions in the Chromium implementation, this provides a reasonably high level of confidence that our implementation is not vulnerable to these attacks.

5.2 Compatibility

To evaluate how much our defense mechanisms disrupts benign functionality of typical web applications, we conducted experiments on a proof-of-concept website we built ourselves and on a broad sampling of existing websites.

The first experiment uses a constructed webpage that contains all the required annotations and third-party scripts. This page functions well in our modified browser. Both advertising networks we tested (Google Adsense and Clicksor) behave normally during testing with no errors even while hiding as much user information as possible from those scripts by marking content nodes as private. Security properties verified previously ensure that embedded third-party scripts cannot access the private content.

For real-world web applications, there is no easy way to automate testing because of the need to create and log into accounts, as well as to interact with the site. This limits the number of sites we can test. We picked 60 sites to test, sampling a range of sites based on popularity. We chose the top 20 US sites according to Alexa.com, 20 sites from sites ranked 80-300 (primarily from 80-100 with some other sites randomly picked from 100-300 to substitute for sites with inappropriate content, e.g. porn sites), and 20 sites from below the rank of 1000 (randomly selected from sites ranked from 1000-10000). For each site, we tested basic functionalities such as login and site-specific operations. These sites contained a variety of third-party scripts including advertising networks (Doubleclick, Adsonar, Ad4game, etc.) and Google Analytics. We isolated the third-party scripts and added the privacy policies on nodes that carry user data. We did not modify the embedded scripts. Policies for these pages are automatically generated by our policy learner which we evaluate more extensively in the following section. Here, we ensure the third-party script identifications are correct. In cases where a compatibility issue arises due to errors in the automatic third-party script identification, we manually correct the policies and test the functionality again. We discuss situations where the policy learner produces an incorrect policy in Section 5.3.

We relaxed our policy learner to always give the <head> tag's write access to third-party scripts. This was necessary since some analytics and ad network scripts inject script nodes in the head region. This does not compromise confidentiality due to the fine-grained nature of our policy: user-sensitive data is never revealed from children

nodes of the <head> tag as long as the tags that directly containing the private information are marked private.

With the assistance of our automatic policy generator and minimum manual annotation effort (mainly helping proxy server to recognize important library scripts as host scripts such as jQuery), 46 out of 60 sites functioned without a problem. Of these, 23 sites do require manual identification of third-party scripts. For example, we added aolcdn.com to aol.com's whitelist as trusted domain.

Of the 14 sites that have problems, four are due to our HTML parser, Nokogiri [16], crashing on the site's HTML. Two sites do not contain login functions, another two sites use only SSL traffic which our current implementation of policy learner cannot tackle. Three sites show significant JavaScript console errors, all due to host script trying to access many guest objects (e.g., _gaq as mentioned before) but our policy learner cannot automatically add the global window object before these accesses. This problem also happened in some other sites, but the access is simple and we can manually add the object easily. For more complicated cases, they can be addressed by either web developer's effort or dynamic modification within JavaScript Engine. Three sites have third-party scripts that try to access a private node and therefore crashed in the process. After a closer look at all these accesses, we found that the private nodes identified by our policy learner are actually all false positives. Appendix A provides more details on the results for each site.

5.3 Policy Learning

To evaluate our automatic policy generator tested requests to sample websites and evaluated the accuracy of third-party script identification and node visibility marking.

Untrusted Script Identification. Our approach marks embedded scripts as untrusted when their origin is different from the page origin. The only false positives we observed resulted from websites that host scripts on another domain. This is fairly common with larger websites. For example, `nytimes.com` embeds some scripts from `nyt.com` which interact with host scripts closely, including accessing variables or functions from host scripts. There is no way to safely infer that scripts from other domains are trusted, so for this situation we resort to requiring developers to manually specify a list of additional trusted domains in the response headers. Scripts from a trusted domain are treated as if they are from host domain and execute in the same world as host scripts.

There are also situations where scripts in the host page appear to come from the host, but should not actually be trusted. This occurs when the host includes a third-party script using cut-and-paste. For instance, Google AdSense and Google Analytics require host pages to include an inlined code snippet. This is safer than an embedded script loaded from the remote site, since at least the host has the opportunity to see the script and knows that it is not vulnerable to a compromise of the remote server, but inlined scripts should still not have access to protected data on the page. Our policy generator has no way to tell whether an inlined script came from an untrusted source. This causes certain functions to break due to the isolated execution environment of two mutually dependent scripts. Our ad-hoc solution is to use heuristics to identify specific patterns in inlined scripts that correspond to commonly inlined scripts. For example, we look for _gat or _gaq in an inlined script and mark scripts that contain them with the

same worldID as the embedded Google Analytics script. Since other scripts may now intentionally add such tokens in their scripts to impersonate *Google Analytics*, this is only a ad-hoc solution. Ideally, service providers would add an appropriate worldID tag in their inlined snippets.

Private Node Marking. To test the marking accuracy of our private node identification approach, we tested the basic functionalities such as login and site-specific operations on the sample sites used for the compatibility experiment. The traffic is redirected to go through the proxy server where modifications are made to the responses including adding ACLs and worldIDs. We recorded the total number of nodes, total number of nodes marked public before login and after login, total number of third-party scripts existing on the page, and the trusted domains needed to be manually added (e.g., scripts stored in Content Distribution Networks and libraries).

Table 3 summarizes the results of our policy generation experiments. Appendix A provides the full details for each tested site. The sample size and ranking denotes the total number of sites we selected from that range of ranking at Alexa.com. $Priv_{NoCred}$ is the percentage of nodes that are marked private based in two identical requests, neither with credentials. Since none of these responses actually contain any personal information, $Priv_{NoCred}$ gives a rough measure of the nodes that are marked as private because they vary between requests even though they do not contain sensitive information. $Priv_{Cred}$ is the percentage of nodes that are marked as private based on normal operation of the proxy. That is, based on the differences between two successive requests, one with and one without login credentials. The last two columns show the total number of third-party scripts on the host page and the number of trusted domains that need to be added to maintain functionality.

There is a reasonable drop in the fraction of nodes that are public after we login to the page, which is exactly what we are expecting. We can also see an increase in public content share after login as the ranking of sites goes lower, which indicates less important sites have less private information.

Statistically, the average number of third-party scripts on a page grows as the sites become less popular. This indicates that less popular sites are more likely to embed untrusted scripts than more popular sites. Finally, the number of trusted domains that have to be added averages less than one per site. This is lower for less popular sites, consistent with the expectation that hosting scripts on alternate domains is more common with popular sites. This result indicates that the effort required for developers to denote trusted sites is minimal.

We also inspected the nodes that were marked as private. Most of them do contain information that most people would consider private such as usernames, email

Table 3. Summary of Automatic Policy Generation Results

Size	Alexa Ranking	$Priv_{NoCred}$	$Priv_{Cred}$	3rd-p scripts	Trusted Domain
13	1-20	28.4%	47.4%	0.8	0.7
11	80-100+	4.3%	21.6%	2.6	0.5
18	1000+	2.0%	17.0%	2.2	0.5

addresses, personal recommendations and preferences. In addition, some nodes that contain session-related advertisements and tags are also marked private, due to values in those tags that vary across requests. These false positives are more frequently seen on the more popular sites, as these sites are more dynamic and complex.

6 Related Work

Much previous work has targeted the challenge of safely executing scripts from untrusted sources in a web page. The two main approaches are to either rewrite the JavaScript code or to modify the browser. An alternative to restricting the private information third-party scripts can access is to move the content-related computation inside the browser, therefore leaking no information at all. This approach is taken by Adnostic [20] and RePriv [5], but since it requires re-architecting the entire web infrastructure we do not consider it further. Here, we categorize previous works by their major mechanisms.

Isolating Execution Environments. Barth et al.'s *isolated worlds* mechanism is designed to protect browsers from extension vulnerabilities [1]. The mechanism they introduced isolates extensions from each other by dividing the JavaScript execution context into several independent ones. We adopt this mechanism to isolate webpage scripts. Since this work do not target privacy, it does not consider mediating DOM accesses.

Adjail [12] is the most similar work to ours. It puts third-party scripts into a shadow iframe with a different domain name, using the browser's same origin policy to isolate that frame and sets up a restricted communication channel between the shadow iframe and host page. This approach does not require any browser modification, but has several limitations including inflexible policies (sub-tree root can only have intersection of children's ACLs), difficulty to accommodate two or more embedded collaborating ad scripts and a complicated, and complex maneuvers needed to preserve impression and clickthrough results.

The HTML5 standard provides a way to execute JavaScript in different threads using *Web Workers* [8]. The goal of this is mainly to improve JavaScript performance by preventing misbehaving scripts from consuming too many resources. Another tag proposed by HTML5 that related to our goals is `sandbox` [7]. This provides some isolation, however the allowed policy is very coarse-grained.

Rewriting JavaScript. ADsafe [3] restricts the power of advertising scripts by using a static verifier to limit them to a safe subset of JavaScript that excludes most global variables and dangerous statements such as `eval`. Caja [14] also restricts JavaScript, but uses an automatic code transformation tool. The rewriting procedure of Caja is very complicated and cannot always preserve original script functionality. Conscript [13] uses aspect-oriented programming to enforce various policies. Its advising functions provide a broad range of policies such as forbidding inline scripts and enforcing Http-only cookies.

Extending Browsers. Jim et al. proposed a per-script policy to defend against XSS attacks [11]. The basic idea is to create a whitelist of the hash of all scripts that are

allowed to run on the page. MashupOS addresses the integrator-provider security gap by introducing several new tags that can be used to restrict embedded scripts in different ways [22]. However, it cannot support the policies needed to handle current ad networks since MashupOS requires the third-party content to be in embedded in a particular way. Following this work, Crites et al. proposed a policy that abandons the same-origin policy by allowing the integrator to specify public and private data including DOM accesses [2]. Completely abandoning SOP would require significant changes to websites. Jayaraman et al. introduced OS protection ring idea to DOM access control [10]. Each node is assigned a privilege level and only scripts within appropriate rings can access that DOM element. Compared to these works, our work has the most expressive policies and easiest deployment.

7 Availability

Our implementation is available under an open source license. The code for our modified Chromium is at `https://github.com/Treeeater/Chromium_on_windows`. The proxy server implementation is at `https://github.com/Treeeater/GreasySpoon-proxy-script`.

Acknowledgments. This work was partly supported by grants from the National Science Foundation and the Air Force Office of Scientific Research under MURI award FA9550-09-1-0539. The authors thank Adam Barth for helpful advice on the isolated worlds implementation. We thank Jonathan Burket, Peter Chapman, Jack Davidson, Yan Huang, and Tianhao Tong for helpful comments on this work.

References

1. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting Browsers from Extension Vulnerabilities. In: 17th Network and Distributed System Security Symposium (2010)
2. Crites, S., Hsu, F., Chen, H.: OMash: Enabling Secure Web Mashups via Object Abstractions. In: 15th ACM Conference on Computer and Communications Security (2008)
3. Crockford, D.: ADsafe: Making JavaScript Safe for Advertising (2007), `www.adsafe.org`
4. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: RFC2616: Hypertext Transfer Protocol - HTTP/1.1, `http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.1`
5. Fredrikson, M., Livshits, B.: RePriv: Re-Envisioning In-Browser Privacy. In: IEEE Symposium on Security and Privacy (2011)
6. The Chromium Development Group. The Chromium Projects: Notifications of Web Request and Navigation, `https://sites.google.com/a/chromium.org/dev/developers/design-documents/extensions/notifications-of-webequest-and-navigation`
7. Hickson, I.: HTML5 specification adding Sandbox attribute, `http://www.whatwg.org/specs/web-apps/current-work/#attr-iframe-sandbox`
8. Hickson, I.: Web Workers in HTML5 standard, `http://www.whatwg.org/specs/web-workers/current-work/`

9. ECMA International. ECMA JavaScript specification, http://www.ecma-international.org/publications/standards/Ecma-262.htm
10. Jayaraman, K., Du, W., Rajagopalan, B., Chapin, S.J.: ESCUDO: A Fine-Grained Protection Model for Web Browsers. In: 30th IEEE International Conference on Distributed Computing Systems (2010)
11. Jim, T., Swamy, N., Hicks, M.: Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In: 16th International Conference on World Wide Web (2007)
12. Louw, M.T., Ganesh, K.T., Venkatakrishnan, V.N.: AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In: 19th USENIX Security Symposium (2010)
13. Meyerovich, L.A., Livshits, B.: ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In: IEEE Symposium on Security and Privacy (2010)
14. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe Active Content in Sanitized Javascript (2007), google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf (revised 2008)
15. Karel Mittig. GreasySpoon, Scripting Factory for Core Network Services, http://greasyspoon.sourceforge.net/
16. Patterson, A.: Nokogiri - An HTML, XML, SAX and Reader parser with the ability to search documents via XPath or CSS3 selectors and much more, http://nokogiri.org/
17. Rogers, M.: Facebook Advertisements Displayed Pictures of User's Friends and Families (2009), http://endofweb.co.uk/2009/07/facebook_ads_2/
18. Singel, R.: Singel-Minded: Facebook comments are another 'Good News, Bad News' proposition, http://www.wired.com/epicenter/2011/03/singel-facebook-empire/
19. Stamm, S., Sterne, B., Markham, G.: Reining in the Web with Content Security Policy. In: 19th International Conference on World Wide Web. ACM, New York (2010)
20. Toubiana, V., Nissenbaum, H., Narayanan, A., Barocas, S., Boneh, D.: Adnostic: Privacy Preserving Targeted Advertising. In: 17th Network and Distributed System Security Symposium (2010)
21. W3C. W3C Document Object Model Level 3 Core Specification, http://www.w3.org/TR/DOM-Level-3-Core/
22. Wang, H.J., Fan, X., Howell, J., Jackson, C.: Protection and Communication Abstractions for Web Browsers in MashupOS. In: 21st ACM SIGOPS Symposium on Operating Systems Principles (2007)

A Automatic Policy Generation Results

Here we present the results from our automatic policy generation experiments. For each site we report:

- $Priv_{NoCred}$ — the number of nodes marked private based on two requests, both without any user credentials (cookies).
- $Priv_{Cred}$ — the number of nodes marked private based on normal proxy operation (two requests, one with and one without credentials)
- 3rd-p scripts — the number of scripts from different origins included in the site.
- Compatibility — any compatibility problems due to our protections.

We group the results into three tables, corresponding to the top (1-50), middle (50-300) and lower (1000+) ranking sites respectively, according to Alexa.com. The result for some sites are excluded due to non-applicability (SSL traffic, no login approaches, e.g.).

Policy Learning Results for Top-Ranked Sites

Sample sites	$Priv_{NoCred}$		$Priv_{Cred}$		3rd-p scripts	Compatibility	Trusted Domain
Google	188/243	78%	209/265	79%	none	none	none
Facebook	25/421	6%	122/195	63%	none	none	fbcdn.net
Yahoo	3417/8273	34%	3600/8221	35%	keywordblocks, s0.2mdn.com	none	yahooapis yimg
Youtube	32/739	4%	494/1195	41%	GA[4]	inline access[2]	none
Amazon	295/1049	28%	640/1490	43%	none	none	images-amazon
Twitter	399/752	53%	100/110	91%	GA	policy violation[3]	twimg,jQuery
Craigslist	0/1045	0%	0/1051	0%	none	none	none
Linkedin	5/262	2%	794/876	91%	GA	inline access	none
MSN	189/676	28%	36/1083	4%	none	none	s-msn.com
Bing	40/187	22%	46/188	24%	none	none	none
Aol	29/703	4%	75/706	11%	player.it	none	aolcdn.com
CNN	4/1416	1%	N/A[1]	N/A	Adsense dl-rms.com questionmarket insideexpressai	none	turner.com
wordpress	26/300	9%	122/349	35%	quantserve gravatar scorecardresearch	inline access	wp.com
Flickr	23/143	16%	663/699	95%	doubleverify s0.2mdn.net	none	yimg.com yahooapis.com

[1] These use scripts to change existing nodes to display user information. No significant number of public to private node changes were detected after login.

[2] Some of these pages' host script try to access vars/functions in third-party scripts and therefore encountered errors. Some of them can be corrected rather easily while others requires web developers effort or dynamic modification in the JavaScript engine.

[3] Some third-party scripts in these pages try to access private nodes. These errors should happen as the scripts violated the policies.

[4] GA stands for Google Analytics.

Policy Learning Results for Middle-Ranked Sites

Sample sites	$Priv_{NoCred}$		$Priv_{Cred}$		3rd-p scripts	Compatibility	Trusted Domain
Twitpic	18/107	17%	43/193	23%	crowdscience scorecardresearch quantserve fmpub gstatic	inline access policy violation	googleapis.com twitter
washingtonpost	1/1722	1%	192/1975	10%	facebook	inline access	none
Digg	33/967	3%	348/1000	35%	diggstatic.com	none	facebook scrorecardresearch
Expedia	66/814	8%	68/814	8%	intentmedia	none	none
vimeo	13/413	3%	229/431	53%	GA,quantserve	none	vimeocdn
statcounter	0/457	0%	53/190	28%	doubleverify	none	none
tmz.com	9/1682	1%	N/A[1]	N/A	quantserve adsonar revsci.net gumgum nexac.com s0.2mdn.net doubleverify	inline access	none
bit.ly	3/105	3%	35/121	29%	twitter,GA	inline access[2]	none
newegg.com	8/1212	1%	10/1212	1%	GA	inline access[2]	none
indeed.com	2/128	2%	9/129	7%	jobsearch,GA scrorecardresearch	policy violation[3]	none
wikia.com	2/417	1%	17/364	4%	GA vimeo quantserve	inline access	none
yelp.com	12/794	2%	115/848	14%	GA	none	yelpcdn
articlebase.com	1/1058	1%	563/729	77%	GA	none	googleapis.com
skyrock.com	427/804	53%	473/865	55%	CDN[4]	none	skyrock.net
btjunkie.com	4/349	1%	1759/2564	68%	Adbrite,GA	none	none
duckload.com	3/158	2%	134/233	58%	GA statcounter	none	googleapis.com

[1] These sites use scripts to change existing nodes to display user information. No significant number of public to private node changes were detected after login.

[2] bit.ly combine jQuery code with Google Analytics, this will not work unless a separate jQuery library is included in the page which is marked as third-party script.

[3] Some third-party scripts in these pages try to access private nodes. These errors should happen as the scripts violated the policies.

[4] Skyrock.com puts many third-party scripts onto their CDN. This is very rare and the scripts are considered first-party because CDNs need to be trsuted.

Policy Learning Results for Lowly-Ranked Sites

Sample sites	$Priv_{NoCred}$		$Priv_{Cred}$		3rd-p scripts	Compatibility	Trusted Domain
gamefaqs	12/451	3%	25/450	6%	i.i.com.com cbsinteractive	inline access	none
timeanddate	1/333	1%	3/333	1%	exponential.com tribalfusion.com	none	none
Armorgames	4/1138	1%	12/1134	1%	GA,ad4game, quantserve	inline access	cpmstar.com
Fantasyleague	13/594	2%	33/580	4%	adtech twitter sumworld	inline access	none
9gag.com	8/529	1%	125/585	21%	Adsense GA	inline access	cloudfront.net googleapis.com
Blinklist	1/321	1%	28/246	11%	GA	none	none
modcloth.com	9/316	3%	17/314	5%	GA	policy violation[1]	none
Getcloudapp	2/51	4%	86/124	69%	GA	none	none
imtalk	10/2096	1%	559/2488	22%	Adsense, addthis statcounter, GA	policy violation	none
change.org	19/355	5%	33/367	9%	GA google map simplegeo quantserve	none	googleapis
url.com	1/262	1%	28/279	10%	GA	none	none

[1] Some of these pages have Google Analytics initialization scripts that does document.getElementsByTagName('script')[0].parent.insertBefore(). Since the first script of the page is not necessarily public, this snippet would fail. Other version of Analytics use document.write therefore does not have this issue.