Alhambra: A System for Creating, Enforcing, and Testing Browser Security Policies

Shuo Tang
University of Illinois,
Urbana-Champaign, IL
stang6@illinois.edu

Chris Grier
University of California,
Berkeley, CA
grier@cs.berkeley.edu

Onur Aciicmez
Samsung Advanced Institute
of Technology, San Jose, CA
o.aciicmez@samsung.com

Samuel T. King
University of Illinois,
Urbana-Champaign, IL
kingst@illinois.edu

ABSTRACT

Alhambra is a browser-based system designed to enforce and test web browser security policies. At the core of Alhambra is a policy-enhanced browser supporting fine-grain security policies that restrict web page contents and execution. Alhambra requires no server-side modifications or additions to the web application. Policies can restrict the construction of the document as well as the execution of JavaScript using access control rules and a taint-tracking engine. Using the Alhambra browser, we present two security policies that we have built using our architecture, both designed to prevent cross-site scripting. The first policy uses a taint-tracking engine to prevent cross-site scripting attacks that exploit bugs in the client-side of the web applications. The second one uses browsing history to create policies that restrict the contents of documents and prevent the inclusion of malicious content.

Using Alhambra we analyze the impact of policies on the compatibility of web pages. To test compatibility, Alhambra supports revisiting user-generated browsing sessions and comparing multiple security policies in parallel to quickly and automatically evaluate security policies. To compare security policies for identical pages we have also developed useful comparison metrics that quantify differences between identical pages executed with different security policies. Not only do we show that our policies are effective with minimal compatibility cost, we also demonstrate that Alhambra can enforce strong security policies and provide quantitative evaluation of the differences introduced by security policies.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design

Keywords

Web browser, web security, cross-site scripting

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

1. INTRODUCTION

Today's web epitomizes the culmination of distributed systems theory and practice, enabling millions of users to access billions of services, scattered around the world, seamlessly and efficiently. Using only a web browser running on a single client machine, users can search the billions of documents on the Web in under two seconds, and modern web browsers can render the slew of buggy HTML documents found on today's Web effectively. Unfortunately, this triumph of performance and compatibility has come at the cost of security. For example, the browser's ability to render incorrect HTML has effectively introduced browser-specific ambiguities into the HTML specification. These ambiguities make it difficult for server-side code to reason about how the browser will interpret data, and is one cause of the overwhelming number of security vulnerabilities, such as cross-site scripting (XSS), in web-based applications. In fact, XSS recently became the most prevalent vulnerability on modern computer systems, accounting for more vulnerabilities than all others combined [22].

Researchers have proposed many techniques for detecting, preventing and containing attacks in web applications. Mitigation techniques can involve the server, the server and the client, or just the client to provide protection to users.

The first, and often most accepted, solution to web application vulnerabilities is simple: fix the bug. Unfortunately, web developers have historically been slow to patch bugs [22], despite the efforts from the research community to make this process easier [3]. Furthermore, recent research has argued that purely server-side techniques are flawed due to differences in browser implementations [16], ultimately limiting the effectiveness of server side techniques.

Hybrid server-client solutions use browser modifications to allow web-application developers to express security constraints to the browser directly. Some recent examples of this type of defensive architecture include introducing new HTML tags for fine-grain sandboxing of scripts [10, 25], HTTP headers to express precisely the provenance of a request to servers [4], or a server-specified whitelist of scripts [10]. Two downsides of hybrid solutions are that servers and clients must both be modified, introducing a high barrier to adoption, and hybrid solutions provide little support for legacy systems.

Client techniques filter attacks and attempt to make pages safe by changing the behavior of web pages [11, 18, 24, 13]. Client side prevention is positioned so that clients can defend themselves against servers even if the servers are malicious or unpatched. Fun-

damentally, scripts execute within browsers, making browsers a natural location to detect and remove malicious scripts, but having browsers change page behavior might affect how the page operates. This potential compatibility issue drives the development of client side mitigation techniques and has caused some designers to deploy conservative designs to "avoid breaking the Web" [18].

To protect users against vulnerable sites we propose Alhambra[1], a browser-based system that can enforce fine-grain security policies and automatically defeat a wide range of web attacks. Alhambra requires no modifications to the server or web application; instead we use a browser to enforce client-side policies that prevent attacks that exploit bugs in web applications, such as XSS. Our approach is to provide a system that reduces attack surface while retaining compatibility. Alhambra has built-in support for monitoring the execution of a web application in the browser and provides support for policies that restrict client-side web application execution. Based on our architecture we have developed two novel policies that aim to defeat XSS attacks.

Alhambra includes a browser-based testing system that enables us to analyze the impact of security policies on the compatibility of web pages. Using browsing sessions generated by people using browsers and stored for analysis enables our system to revisit each web page using different security settings quickly and automatically. To provide comparison between policies for identical pages we have also developed useful comparison metrics for quantifying differences between identical pages executed with different security policies. Our metrics operate at the visible level as well as examining execution and parsing information generated by our browser.

We have developed two policies that use the Alhambra architecture for enforcement and testing. Our first policy is designed to prevent DOM-based XSS attacks or detect potential vulnerabilities, where client-side code is unsafely using untrusted input. This policie leverages an object-level taint tracking engine that we build into the browser to identify insecure use of untrusted content. Thus, Alhambra can either prevent those content from being executed as JavaScript code or report a potential DOM-based XSS bug in the web application.

The second one uses information generated by visiting and analyzing web applications to automatically generate policies. Automatically generated policies are based on page structure, and are designed to prevent content injection attacks, such as XSS attacks. During a brief training phase, we learn the structure of a page, and then during subsequent visits to this page we enforce this structure, eliminating content injection attacks that perturb the structure of a page.

We show that Alhambra is able to prevent real attacks and evaluate degrees of compatibility for sophisticated web applications. Using browsing sessions provides the information required to test policies automatically and we show that our policies have small or no impact on web browsing. For the policies we present in this paper, we see negligible impact on the parsing, rendering, or execution of complex web applications.

To the best of our knowledge the contributions of this paper are as follows:

- We present a novel browser architecture with built-in mechanisms for enforcing browser security policies.
- We present a browser-based testing system that uses user-generated browsing sessions to enable policy testing.
- We show that using object-level taint tracking in the browser can prevent DOM-based XSS vulnerabilities.
- We show how by using only information available at the client we can enforce policies that can prevent a wide range of content injection attacks (i.e. XSS attacks), without changing the user-visible operation of the site.

[1]Mosaics pervade Islamic art, but many of the great mosaics have been destroyed. One of the best preserved collections of mosaics has been protected at the Alhambra fortress in Cordoba, Spain.

2. ALHAMBRA ARCHITECTURE

This paper describes Alhambra, a browser-based system for creating, enforcing and testing browser security policies. We have three main goals that drive the creation of useful browser security features. Our first goal in Alhambra is to develop techniques that allow browsers to automatically defend themselves from attack. Our second goal is to avoid causing unreasonable incompatibility and to permit as much of existing web application functionality as possible. Our third goal is to run policies client side without any help from the server.

In this section we describe the design of the Alhambra browser. We first discuss our overall architecture for preventing attacks (Section 2.1). Then we describe how Alhambra enforces policies on the structure and execution of web applications (Section 2.2).

2.1 Alhambra

Alhambra consists of two components: a policy enhanced browser and a replay system for testing (Figure 1). At the heart of Alhambra is a browser capable of enforcing fine-grain security policies. The browser uses a policy layer to impose restrictions on the execution of each page. Security policies are completely client-side and require no server support in order to be enforced. The browser has the unique responsibility of parsing and executing scripts and is the final authority that can remove malicious scripts before they are executed. By positioning security policy and enforcement mechanisms inside the browser we enable the browser to prevent attacks.

An important and often overlooked aspect of attack prevention is the compatibility cost of deploying security policies. Compatibility can be at odds with security and Alhambra must have minimal impact on benign pages to be acceptable to user browsing the web. To determine if our security decisions impact the functionality of a web page we have constructed Alhambra to include testing functionality and metrics for measuring compatibility. Figure 1 shows how the browser and replay engine are used to test security policies. By testing each security policy we develop and combining successful policies we can choose the right policy for a site. By further testing each policy with our framework we are able to identify compatibility problems quickly and automatically.

2.2 Enforcing policies

In general, the Alhambra system is well-suited for a wide range of client-side policies, so we chose policy enforcement properties based on two key goals. First, we want to hone in on fundamental properties to client-side security policy to provide strong protections. Second, we want to offer flexibility to implement a variety of policies in order to test and evaluate the impact on compatibility of each policy.

Alhambra enforces policies as the page is parsed and scripts are executed. There are two primary locations where policy can be enforced: document structure and execution. Structural policies can restrict the creation of elements or place limitations on the type of content in a web page, policies of this type are enforced on interactions with document tree. Policies can also restrict the execution of scripts by disabling methods or restricting access to methods

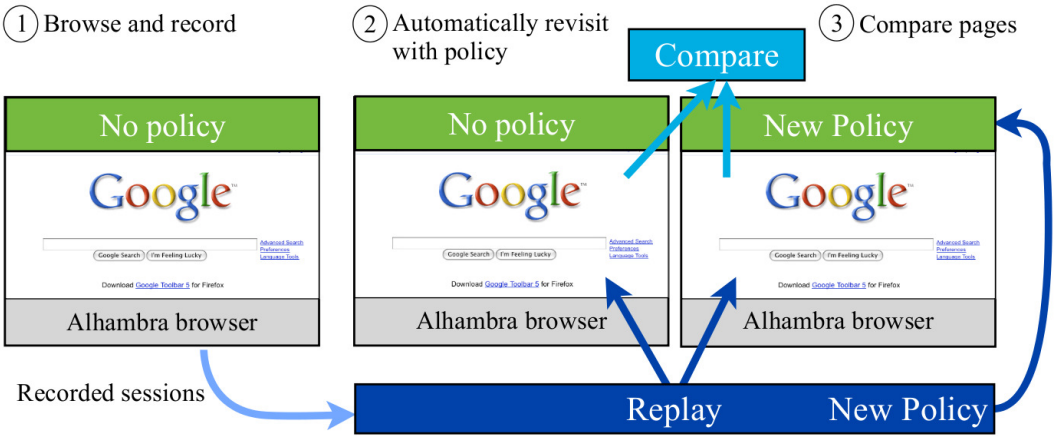

Figure 1: The Alhambra system uses replay to enable policy testing of identical pages. Alhambra captures and records a browsing session using the replay engine, and plays back to instances of the Alhambra browser with a policy and without a policy. Comparisons on the content are then performed to evaluate compatibility.

based on runtime information. Both types of policies are enforced throughout the lifetime of a page within the browser.

To enforce policy during execution, we have built an object-level taint tracking engine into Alhambra, allowing policies to specify execution constraints based on the flow of data within the client-side portion of a web application. We have developed policies that seek to prevent DOM-based XSS attacks and detect DOM-based XSS vulnerabilities and present them in Section 4.2. In Alhambra taint-tracking is built into the JavaScript engine and HTML engine and allows taint information to be propagated as data flows through JavaScript runtime and DOM tree. Policies can use the taint-tracking engine to control execution by specifying methods that cannot be allowed to execute on tainted objects. This mechanism is described in greater detail with the policy in Section 4.2.

In addition to page execution, we observe and enforce the *document structure* of web pages. The concept of document structure was introduced by Nadji et al. [16], and refers to the parsed tree generated by the parsing of a web page. In the web browser the HTML parser generates a document tree from the text contents of a page. Every page contains an HTML document composed of different types of HTML elements (<img>, <p>, <script>, etc...). The W3C Document Object Model (DOM) standards [1] provide a public reference for browser interfaces to elements in a web page. Our definition and use of document structure differs from the term introduced by Nadji et al. since we combine both dynamic document structure and static document structure.

In Section 4.3 we describe a document structure integrity (DSI) policy that uses browsing history to create rules that limit the document structure of a page. Generally, our policy engine allows rules that prohibit elements within a particular document. For example, a policy can prohibit any non-inline scripts and only allow scripts from a set of whitelisted domains (similar to part of the Mozilla Content Security Policy [15]).

3. TESTING POLICIES

Alhambra uses browser support for testing browser policies. Since policies can modify the functionality of the page, our intent when testing them is to determine if the look and functionality of that page are impaired. To test policies, we develop a browser-level system capable of revisiting user generated browser sessions, and we propose three metrics of comparison to determine if our policies make unwanted modifications.

Alhambra is the first system to include the ability to perform side by side comparisons of browsers and isolate the effects of security policy on compatibility. Testing the impact of security techniques can be difficult and we provide a systematic method for testing the compatibility of browser security techniques. A fundamental aspect of testing security policies is the ability to allow policies to modify the execution and contents of the page while the system is revisiting a browsing session.

This section describes the design and implementation of our browser-level mechanisms for testing compatibility of our security policies. First, we discuss a naive testing technique and why it is insufficient for testing security policies (Section 3.1). Next, we describe our browser-level revisiting mechanisms and how they overcome the shortcomings of the naive approach (Section 3.2). Finally, we discuss our techniques and quantitative metrics for comparing two pages rendered using different security policies (Section 3.3).

3.1 Naive policy testing

A naive approach to testing a policy consists of three simple steps. First, the browser could navigate an unmodified browser and a policy-enhanced browser to the page to be tested. Second, the browser could replicate user actions such as typing, clicking and other interactions in both browsers. Third, the browser could compare the resulting pages in both browsers by inspecting the rendering and functionality.

This naive approach is very easy to deploy and the only requirement is browser support for replicating user actions, which could be implemented using a variety of techniques inside or outside the browser. Though easy, different sources of non-determinism through page execution make the naive testing system undesirable as the main method of testing policies.

A single page viewed multiple times can differ due to server-side and client-side non-determinism. Server-side non-determinism is caused by the web server providing different content for identical requests. The content returned to the unmodified and policy-enhanced browsers could change based on the time of day, client or server state, or other dynamic decisions even if both browsers request identical URLs. Moreover, even if the two browsers get the same web pages, they could make different subsequent re-

quests because of client-side non-determinism. For example, a web page could use a random number to choose which advertisement to download.

Finally, re-issuing non-idempotent requests, such as HTTP POST requests, may create unanticipated and undesirable side effects. For example, a user might buy a book at `amazon.com`. If we wanted to test this action with ten different policies, the browser might end up purchasing ten extra copies. Furthermore, making HTTP GET requests idempotent is considered a good practice, but unenforced in modern web systems, making it nearly impossible for the browser to reason about which requests will induce undesirable side effects.

3.2 Testing policies in Alhambra

To overcome the shortcomings of the naive approach, we developed a browser-based testing system designed to recreate past browsing sessions to test security policies. Using our system we are able to test policies quickly and accurately to determine the compatibility impact of our security decisions.

Logging and replay are widely used for replay of programs. Generally speaking, the characteristics of browsers and web applications determine the type of information that needs to be logged for re-execution. We built Alhambra by modifying the OP web browser which uses a message passing interface for communication between browser components [8]. Non-deterministic events that need to be recorded fall into two categories: messages from the browser kernel received by web applications and other external inputs. OP's architecture provides the infrastructure to log messages between processes and enables recording of network replies, storage operations (both cookies and file system) and user actions. In this section we describe the challenges we face when applying the high-level ideas from existing replay systems [14, 21, 7, 6] into a new application, namely, a web browser.

Execution order: After applying new policies, it may be impossible for the browser to execute the same as the initial recording. During re-execution, our primary concern is the output of the web application. To evaluate policies we do not need to constrain the execution path and allow the execution of the web application to flow freely.

Missing network requests or function calls: By labeling network replies with the corresponding URL we can skip replies associated with unrequested URLs during page execution. Missing functions are more difficult. To illustrate the difficulty, assume we have two script fragments S1 and S2 and they both include a call to the random number generator (i.e. <script>random()</script>). If a policy removes S1, we cannot decide which logged value to give when random in S2 is called. One solution is to label the returned value with the location of the script element so the browser can return results for the correct instance of the function call.

Extra network requests or function calls: While less common than the previous cases, it is possible to encounter causal dependencies in scripts and HTML. An example of a script that could cause new network requests or function calls to be seen during re-execution is shown in Figure 2. If the policy removes an element that is later used for calculation, the resulting function calls or network requests could differ from the recorded session. For new network requests we can either return blank content or let the network component fetch a new one. Function calls can be dealt in a similar way to network requests. Either choice indicates that the policy has introduced differences in the execution of the page and this should be used to evaluate our security policies.

User actions: During testing, a policy could remove the target of user actions. If the target of user action is removed, we attempt the user action anyway and capture the differences resulted in execution. Although the user action could result in new execution, the presence of these new actions does not invalidate our indent of testing policies.

```
if(iframe1) {
    iframe1['src'] = a.com;
    foo();
} else {
    bar();
}
```

Figure 2: JavaScript statement that can cause different requests to be seen during testing. The removal or alteration of the document can cause different calculations and result in new code being executed.

User interaction with the browser can present challenges when testing the execution of web pages. For each user action, the target of the action must be in the correct state before the user action will have the desired effect. For example, when performing a mouse click on top of a button, if the button has not yet been created the click has no effect. Fortunately, the browser kernel in OP records all browser level messages and provides total ordering on the message log. This guarantees that if a user action is sent to the web page, the target has already been generated since the network message containing the page content has already been delivered.

Our browser-level testing system does have some limitations. For example, we cannot deal with re-ordering of non-deterministic function calls, such as random(), inside the same script. Though it is possible to demonstrate cases where this causes problems for our testing system but we have not encountered it during our testing.

3.3 Comparing browser instances

One approach to comparing browser instances could be to use strict equality testing where we verify that browser instances with and without policies are identical both in page structure and overall behavior. Although it is desirable to have absolutely no impact on benign content, it is possible that our policies might impose overly strict limitations. However, page equality can often be too unforgiving and modifications to the page by security policies could still allow the page to remain completely functional from a user's perspective.

We have developed three metrics to determine if functionality is lost given that our policy has made a modification to the page. The first method uses automatic image processing techniques to identify graphical differences in the rendering of pages. We use the scale-invariant feature transform (SIFT) [12] to identify keypoints in the image and match keypoints between images. SIFT is used for many different image processing applications such as object recognition and stitching. The result of the SIFT keypoint matching algorithm is the number of keypoints that match between two images. We establish a threshold to use for comparison by using SIFT matching on screenshots of identical web pages.

The second metric examines differences in the parsed document tree. If the policy removes an element, the resulting document has obvious differences – the elements removed. Document tree comparison can be done strictly, requiring identical leaves in both trees, or loosely, by comparing the structure and only require the type of each leaf to be the same.

Monitoring and comparing network requests provides a third metric for comparing and evaluating the impact of policy on page execution. During page replay we provide synchronized net-

```
<HTML>
  <HEAD><TITLE>Hello</TITLE></HEAD>
  <BODY>
    Hi
    <SCRIPT>
      var pos =
        document.URL.indexOf("name=")+5;
      var len = document.URL.length;
      var name =
        document.URL.substring(pos,len);
      document.write(name);
    </SCRIPT>
  </BODY>
</HTML>
```

Figure 3: A simple web application with DOM-based XSS vulnerability.

work and file system state and by recording network requests and browser persistent state changes we can additionally monitor the execution of web pages.

4. BROWSER SECURITY POLICIES

Using Alhambra we have developed two policies designed to prevent cross-site scripting (XSS) attacks. In this section we first briefly describe XSS attacks (Section 4.1), then our policy that uses Alhambra's taint-tracking engine to prevent a class of attacks that exploit bugs in the client portion of a web application (Section 4.2). Our second policy (Section 4.3) limits the contents of a document by observing pages in the browsing history to further limit the potential for attack. We present the results of testing both policies in Section 5.

4.1 Attack background

We have designed policies to specifically target attacks that originate on web pages and do *not* exploit the browser, but instead leverage browser capabilities to execute attacks against the client or server. We assume that in general, the authors of the web site are good-intentioned, though there are bugs in the web application that allow an attacker to inject content into the page. XSS vulnerabilities allow an attacker to inject content that is executed as JavaScript by the client.

An XSS vulnerability is a bug in a web page that allows an attacker to inject content into the victim page that is then provided to a browser and executed. Typically XSS vulnerabilities permit the injected code to run with the same permissions as any other code on the page, allowing the attacker's code to access the victim's cookies, page content and any other resources protected by same-origin policy. There are three classes of XSS attacks categorized based on the method that they are included into the victim page. Reflected attacks rely on data provided by the browser being inserted into the page. Reflected attacks are transient as the server does not store the attack code, but provides it back to the browser. The second class is a stored XSS attack where the server embeds the attack code into the page requested by the client but the attack code is stored on the server and provided independent of the request data. Stored attacks commonly result from user-created content provided by a web application. The third class of XSS attacks are DOM-based attacks and are similar to reflected attacks except that they exploit a problem in the scripts included by the page rather than scripts on the server.

In Figure 3 we show a simple web application with DOM-based XSS vulnerability. In the example the JavaScript parses the URL to obtain the visitor's name and then embeds the name into the page's HTML. Assuming this web application is hosted at `http://www.domxss.com`, a URL such as `http://www.domxss.com/#name=Alice` would work as developer intended and print "Hi Alice" in the web page. However, an attack can be constructed using a URL such as `http://www.domxss.com/#name=<script> alert(document.cookie)</script>` causing the web application to write the script tag into the document and then execute it. An exploit such as this can be used by an attacker by tricking users into visiting the malicious URL.

DOM-based XSS vulnerabilities can be far more complex than the example; however, the root cause is that a web application's client-side logic insecurely uses objects that can be controlled by attackers. Objects available to the attacker include the one used in the example, `document.URL`, and without careful sanitization these objects can lead to unintentional JavaScript execution, resulting in an XSS vulnerability.

Malicious payloads that exploit DOM-based XSS vulnerabilities are not always sent to server making it infeasible to detect them in the server side. Web application defenses are also difficult, as they require the web application developer to add sanitization functions throughout their code. Even if web developers write correct sanitization functions it is still possible to miss some cases that result in JavaScript injection. To mitigate the risks of DOM-based XSS attacks we use taint-tracking in the browser to prevent XSS vulnerabilities.

4.2 DOM-based XSS prevention

As we described in Section 4.1, in a DOM-based XSS attack a malicious payload is injected into the vulnerable web application during execution by the browser. To prevent DOM-based XSS attacks, we have implemented data flow tracking in the browser and developed suitable policies in Alhambra to prevent untrusted input from being executed as JavaScript.

4.2.1 Taint-tracking

In Alhambra, objects originated from untrusted sources are marked as tainted and taint information is propagated as the web application interacts with tainted data. Alhambra propagates taint information at the JavaScript object level and we have extended both JavaScript and HTML engines to support taint-tracking. In the JavaScript engine, we have added a field to every JavaScript object to indicate if an object is tainted. To propagate taint, we handle three types of operations on JavaScript objects: assignments, logic or arithmetic operations, and string manipulation. In an assignment, the left operand becomes tainted if the right operand is tainted. For logic or arithmetic operations, the result is tainted if any of the operands are tainted. For string manipulation, the resulting string is tainted if it contains content from tainted sources. For example, any substring of a tainted string is tainted as is the lower case conversion of a tainted string. Similar to Yip et al. [29], Alhambra does not track of implicit data flows.

In addition to the JavaScript engine, Alhambra also needs to track data flow inside the HTML engine since JavaScript objects can be stored in the DOM tree and later retrieved by other JavaScript. To solve this problem, Alhambra taints the DOM nodes where tainted JavaScript objects are stored and upon retrieval, taints the JavaScript objects that interact with the tainted DOM node.

4.2.2 Prevention policy

Policies can specify restrictions on the execution of taint within the browser using the taint-tracking capability in Alhambra. In this section we present a policy designed to prevent DOM-base XSS attacks in web applications.

To prevent DOM-based XSS attacks, DOM objects that can be controlled by an attacker are marked as tainted. Objects that we consider tainted are: `document.URL`, `document.referrer`, `document.location`, and `window.location`. This policy forbids the JavaScript engine from executing tainted input by using propagated taint information. If JavaScript source is constructed from tainted DOM objects, the interpreter will refuse to execute the tainted input. Using the example shown in Figure 3, a URL containing a script payload (`http://www.domxss.com/#name=<script>alert(document.cookie)</script>`) will result in the script tag written to the page being tainted. When the JavaScript engine is invoked to execute the contents of the script tag, our policy prevents it from executing. In Section 5 we present the results of evaluating this policy using the Alhambra testing framework and show that this policy is able to prevent all the attacks we examined while not introducing any incompatibilities in other web applications.

4.3 Automatic document structure integrity

The automatic document structure integrity (DSI) policy is created using information generated by visiting and analyzing web applications automatically. Visits to web pages are used to automatically generate policies based on page structure and behavior, and are designed to prevent content injection attacks, such as XSS and CSRF attacks. During a brief training phase, we learn the document structure of a page, and then during subsequent visits to this page we enforce this structure, eliminating attacks that perturb the structure of a page.

This section describes the key design decisions we had to make when developing policies based on document structure for Alhambra. First, we had to choose at what level of granularity to enforce our policies Second, we had to use properties that will allow us to converge on a policy quickly.

4.3.1 Policy granularity

Our policies must be fine-grain enough to prevent a wide range of attacks, and be coarse enough to withstand updates to page content. To achieve this balance we specify policies that only restrict potential sites of attack and allow the majority of the page to execute unhindered.

Our policy generation tools focus on elements that have source attributes or are easy for attackers to inject or leak information from a victim site. Plugins, images, scripts, and iframes are examples of elements that are commonly injected or used to exfiltrate information from a victim site. Our policies restrict document structure in such a way as to prohibit attacks from using these attack vectors. Specifically we target elements with "src" attribute and inline scripts, and Alhambra enforces their position within the structure of the document.

We do not require specific structure on other elements, formatting, or fonts, and purely cosmetic segments of the page are unrestricted by security policy. This enables our policies to withstand updates by web application developers, which we discuss in Section 5.1.2. Large structural changes, such as migration to a content delivery network, or if the company were to transition to new infrastructure, would likely introduce changes that our techniques cannot accommodate.

4.3.2 Policy convergence

The browser must be able to provide a high level of security for sites while introducing minimal compatibility problems. By using browsing history, the browser can quickly analyze and provide security policies for previously unknown sites. The tradeoff in the speed at which the browser can decide on a policy is directly related to the diversity of structure on site and impacts the compatibility of our policies on pages. In general, converging faster provides a more restrictive configuration with potentially stronger security, but potentially less compatibility since portions of the site will be unexplored.

Alhambra aggregates policy data for all pages at sites with the same second and top level domains (for example `portal.acm.org` and `www.acm.org`) to more quickly converge on a policy. In Alhambra it is possible to specify policies at many different levels, including using the origin or the full path of a URL. Automatic DSI policies are generated using data from all pages seen at a particular second and top level domain, this policy is then applied using the same filter on the domain.

When Alhambra encounters elements with "src" attributes, we generalize each time identical structural elements are seen for different "src" attributes during the training phase. For example, after seeing an image at `sub.a.org` and `www.a.org` we can combine these to allow images from `*.a.org`. The risk in this type of generalization is that we could generalize too much, opening the system up to potential attacks.

One possible enhancement to our overall policy convergence technique is that large structural changes can be seen by monitoring the execution of the policy and noting consistent and frequent violations of the security policy. Using results of the policy violations we can incorporate changes over time into our policy.

A second possible enhancement is to aggregate data for a set of users, rather than a single browsing session. This would provide immediate protection for new sites if other users have contributed policy data. However, the result is a wider attack surface and a policy that could allow more functionality than is typically used by a particular individual.

5. EVALUATION

Alhambra is based on a modified version of the OP web browser running on top of Linux. OP uses WebKit [26] to handle JavaScript and HTML and is written in C++. We have instrumented WebKit to provide our policy engine with the information required to evaluate and enforce structural and behavioral policies. Our browser-based testing system enhances the auditing logs from OP with additional information to revisit pages from past browsing sessions.

All experiments were conducted on a 2.66GHz Intel Core 2 Duo with 8GB of memory and a 250GB serial ATA hard drive. The OS is 64 bit Fedora Core 10, running Linux kernel 2.6.27.

In this section we first present our use of Alhambra to test the policies in Section 4.2 and Section 4.3 using a few popular web applications. Then we test the security policies we have developed using Alhambra against previously reported XSS attacks documented by the XSSed.org project as well as DOM-based XSS attacks in the wild. We also provide overall system performance.

5.1 Testing policies

We have used Alhambra to test both automatic DSI policies and the taint-tracking policy for popular sites that we use on a daily basis. For sites that offer different services to users with accounts we either create accounts for testing or use our personal accounts. We test the policies with two different use cases: basic functionality and interacting with the web application. Basic functionality tests

Site	Test	DOM	Img	Net
www.facebook.com	basic	0%	Pass	2%
www.facebook.com	interactive	1%	Pass	4%
docs.google.com	basic	0%	Pass	9%
docs.google.com	interactive	0%	Pass	0%
en.wikipedia.org	basic	0%	Pass	0%
en.wikipedia.org	interactive	0%	Pass	0%
sfbay.craigslist.org	basic	0%	Pass	0%
sfbay.craigslist.org	interactive	N/A	N/A	N/A
www.cnn.com	basic	0%	Pass	8%
www.cnn.com	interactive	5%	Pass	9%
www.amazon.com	basic	1%	Pass	0%
www.amazon.com	interactive	0%	Pass	6%

Table 1: Results from testing the automatic DSI policy using Alhambra. Percentages indicate the measured difference between replay with and without policy, zero percent is the ideal case. For image comparison, a pass indicates that the SIFT matching indicates they are the same page.

pick simple pages at each site and tests to make sure policy does not impede viewing the page. Interaction involves user input, such as typing a message or editing a document and includes navigation from one page to another. We use recorded browsing sessions and use the Alhambra testing framework to revisit each of the pages in the recorded session.

5.1.1 Web application tests

Facebook. As our first case study we choose Facebook, a popular social networking site. Facebook allows users to create profiles, applications, and in general edit the content that Facebook profiles display. To test basic functionality using Facebook we use sessions that login and browse the site and perform basic actions such as accessing profile pages. Our test for interacting with the Facebook application performs a few common actions. First, we change the status message for our Facebook account. Then we post comments to friends' updates and check notifications.

Google. Google offers a number of services beyond the search portal that offer functionality similar to that of desktop applications. We login to Google and test the Google Docs document editor. To test basic functionality we open a document previously saved. Our test for interacting with Google Docs demonstrates the ability to write a text document. In our test, we create a new document and compose a short paragraph, save it and then use the Google provided UI to format text in the paragraph.

Wikipedia. Wikipedia is an online encyclopedia that allows anyone to edit and contribute to articles. Basic use of Wikipedia is simple and involves accessing Wikipedia articles. We access the special random page that directs the browser to a new random page at `en.wikipedia.org` as well as loading the main page. Our test for interacting with Wikipedia tests the ability to edit an article. We choose a random document, edit the contents and preview the modified document.

Craigslist. Craigslist allows users to view and post ads with very simple markup and formatting. Craigslist has different sites based on geographic location and is similar to newspaper classified ads. The test for basic functionality at Craigslist uses the browser to browse advertisements in the San Francisco bay area (`sfbay.craigslist.org`). To test interaction with Craigslist, we search for and then post an advertisement.

CNN. CNN is a popular news source that hosts free online content and provides different types of content such as movies, images and articles. The test for basic functionality at CNN involves accessing news articles by following a link on the main page. CNN is not quite an interactive website. For interactive test, we look up local news.

Amazon. Amazon.com is online store that sells virtually anything. For testing basic functionality at Amazon.com we browse to a product page. Testing interaction at Amazon.com involves searching for a product and initiating the checkout process, though to prevent many unwanted items arriving in two days or less, we do not complete the checkout process.

5.1.2 Policy compatibility

The remainder of this section presents the compatibility results for our policies using the user-generated browsing sessions. The compatibility results for each site tested are presented in Table 1. For each comparison we provide the percent different from an unmodified replay of the identical page. For visual differences, we compare the rendered web pages using the SIFT algorithm and compare the number of matched keypoints against the values for a set of training data generated by the same sites. For DOM differences, we calculate the edit distance between the two DOM trees. For network differences, we divide the number of omitted network requests by the total number of network requests. The comparisons are generated once the page has reached a steady state and after the interactive session has completed.

DOM-based XSS prevention. The taint-tracking policy that prevents DOM-based attacks (Section 4.2) does not cause any compatibility problems on these pages, and we have additionally tested this policy for the top 100 sites ranked by Alexa [2]. Table 2 presents the results of testing this policy. On the top 100 sites we also see no incompatibilities as a result of enforcing our taint-tracking policy; however, our policy is configured to generate warnings when tainted data is passed to the HTML parser and could result in a DOM-based XSS attack. Fourteen sites generate warnings, one of which we constructed an attack to exploit the vulnerability found. This vulnerability is in a popular website[2] and can be used to inject arbitrary JavaScript into the page. After generating a sample attack we confirmed that our policy prevents the attack.

Automatic DSI. The automatic DSI policy (Section 4.3) limits the structure of pages and will modify some pages slightly by removing the elements that violate the automatic DSI policy. For each of the browsing sessions tested in Table 1 the automatic DSI policy does not introduce any rendering incompatibilities and overall document tree differences are less than 5%. The network has slightly higher differences, in the case of `www.cnn.com` 9% of the network requests differ after the automatic DSI policy is introduced. The network connections that differ due to the policy are due to some included files conflicting with the automatic DSI policy and do not cause the page to function differently. The results of the interactive test for `sfbay.craigslist.org` show incompatibilities. The session being revisited is unable to complete since the policy introduces slight differences in the rendering of the page. These small visual differences result in the final user action (clicking to post an advertisement) missing the button on the page.

Effect of page updates. We also use Alhambra to test the effect of page updates on the compatibility of our automatic DSI policies as the document structure of a page can change due to site updates or other server-side configurations. We use `archive.org` and record browsing sessions that visit pages from over one year ago. Each of the browsing sessions is then revisited to determine the impact on page updates. Unlike our previous tests, we do not have in-

[2]We have disclosed this vulnerability to the site administrators but have not been contacted back.

Site	False positives	Prevented
Amazon	0	N/A
CNN	0	N/A
Craigslist	0	N/A
Facebook	0	N/A
Google	0	N/A
Top 100 sites	0 (14)	N/A
DOM-based XSS attacks	N/A	10/10

Table 2: The taint tracking policy prevents all of the attacks found and has zero false positives on our test set and the top 100 sites reported by Alexa. Fourteen warnings are generated indicating that tainted data was written to the web page but not executed by the JS engine.

Site	Reported	Prevented	Percent
Amazon	4	3	75%
CNN	11	8	73%
Craigslist	20	20	100%
Facebook	11	9	82%
Google	9	8	89%

Table 3: The automatic DSI policy prevents many of the XSS vulnerabilities reported for the sites shown. The reported XSS column is the total XSS attacks examined and the prevented column is the number prevented by automatic DSI policy.

teractive browsing sessions for these pages since `archive.org` does not run the server-side components to the web applications. We found that all of the automatic DSI policies we generated introduce no more changes than we found on current pages (Section 5.1).

5.2 Security analysis

DOM-based XSS prevention. We have tested our taint-tracking policy against publicly disclosed attacks and examples of DOM-based XSS attacks. Table 2 shows the results of testing this policy. We test against five documented XSS attacks disclosed for popular services. We also generate five synthetic attacks based on examples used to demonstrate DOM-based XSS attacks. For both sets of attacks our policy successfully prohibits the execution of the injected script.

Automatic DSI. We have evaluated our automatically created document structure policies against a number of previously documented XSS attacks by XSSed.org [27]. Table 3 presents the results of our evaluation. For each site we examine the most recent reported XSS vulnerabilities and test the vulnerable page using our automatically generated policies for each site. A few reported attacks for each site failed to demonstrate an exploit even on an unmodified browser and those have been removed from the data set. There were no archived attacks for `en.wikipedia.org` on XSSed.org so we have omitted it from Table 3. Additionally, the XSS vulnerabilities reported for Craigslist all exploited the same bug but at different subdomains.

Our policies can prevent many forms of XSS and inclusion attacks though there are still possibilities for attackers succeed. Our automatic DSI policy does not inspect the contents of script elements and an attacker can insert malicious content that mimics the same document structure as the benign pages. One of the Facebook XSS attacks that was not prevented, injected an inline <`script`> into the header, a location where the policy allowed script elements.

5.3 Performance

To evaluate the performance of Alhambra, we isolate the effects of the taint-tracking modifications and structural policy enforcement. To support taint-tracking policies the only overhead added is the manipulation of the extra field to propagate taint in JavaScript objects or DOM nodes. Our performance tests indicate that there are no measurable latencies added to support taint tracking.

Enforcing document structure policies has two different points where latency can be introduced. The first is during a change to the document causing the structural policy to be checked. The second is during JavaScript accesses to DOM methods and properties. We add no measurable overhead for the applications we have tested except for Facebook, which adds around 2x overhead during the parsing stage and none during execution. Most of the overhead incurred by enforcing policies is seen during the parsing phase when the document is built because for each addition to the document tree, the HTML engine checks to ensure that the addition does not cause a policy violation. In our current implementation we check the entire document each time. A simple optimization would be to check if *only* the added element violates our security policy, reducing the overhead significantly.

6. RELATED WORK

The closest work to ours are other systems that use testing and replication to detect problems with security policy. Doppleganger [20] is one such system that applies fine-grain policies to cookies and uses a parallel browser for backup when a cookie policy is unknown. When Doppleganger does not have a cookie policy, two browsing sessions are maintained, one with a restrictive cookie policy and one without. If differences in the page are detected the user is prompted to choose the desired functionality. Doppleganger uses a simple replay system when not mirroring the session to generate the page. In contrast, our reply system removes server and client-side non-determinism and uses replay to detect modifications made by security policies.

XSS prevention. There has been extensive research in preventing and detecting XSS attacks. In a recent work, Nadji et al. use document structure information provided by the server and client side taint-tracking mechanisms to enforce the provided document structure integrity properties during page execution [16]. Similarly, we use concept of document structure to mitigate attacks though we are able to do so by determining the structure automatically and without added server help. Our techniques also require no changes to the contents of pages.

NoScript [13] is a Firefox extension that includes limited XSS prevention and other useful security features. The current version of the extension supports automatic prevention of many reflected XSS attacks by inspecting and sanitizing parameters in the URL. In addition to reflected XSS attacks, NoScript has the potential to block almost any XSS attack since it can selectively enable and disable JavaScript for parts of a page though this must be configured manually. Like NoScript, the IE 8 XSS filter performs heuristic matching on URL contents to prevent reflected XSS attacks.

Different approaches for client side prevention include filtering, taint tracking and content blocking. Noxes [11] is a client-side proxy that filters outgoing network requests based on a dynamic white list assembled for each page. Noxes primarily targets information leakage attacks and could experience high incompatibility due to the aggressive denial of dynamically created network requests. Vogt et al. use client-side taint-tracking to identify requests that leak sensitive information across domain boundaries [24]. Their technique works inside of the browser and pro-

vides security alerts when sensitive information is sent to a third-party domain.

Other techniques have been developed for server-side XSS prevention and modify the page so that it can be delivered safely to the client. Blueprint [23] provides a safe method for delivering content to browsers using server side techniques and JavaScript. By delivering unauthorized content to browsers in a manner that prevents JavaScript execution XSS attacks can be defeated. XSS-GUARD [5] employs a browser framework on the server-side output to identify and remove malicious scripts that are not intended by the web application. Xu et al. use taint-tracking to provide security policies with added information to make server-side decisions [28]. Using taint-tracking, the authors show that they are able to defeat attacks using their techniques. These techniques all operate using server-side additions to provide security and with some modifications could be used in addition to Alhambra to provide stronger attack prevention.

BEEP [10] and Noncespaces [9] are hybrid methods that modify both browser and web server to provide resistance to web application attacks. Noncespaces is able to differentiate between untrusted and trusted content in pages using XML namespaces. Noncespaces modifies the web application to use XML namespaces and provides a policy to the browser restricting XHTML. BEEP uses policies to determine which scripts should execute and the browser consults with server and provided policy when executing to ensure that attackers are unable to inject scripts. These approaches have similar goals to ours, however, Alhambra does not require server modifications and uses document structure to restrict pages.

Taint-tracking or data flow tracking has been used extensively to improve application security. To the best of our knowledge, Alhambra is the first one to use taint-tracking to prevent DOM-based XSS attacks. Existing techniques such as Perl supports a taint-mode [17] to prevent unsafe use of untrusted input. Sekar [19] provides an effective and language-independent taint-tracking approach to prevent injection attacks. Resin [29] is another tool that tracks data flow to propagate policies to improve application security. However, Resin only requires that untrusted data be sanitized to prevent XSS vulnerabilities and assumes the correctness of the sanitization.

7. CONCLUSION

We have presented a client-side architecture for the enforcement, creation and testing of browser security policies. Alhambra demonstrates two new policies that successfully prevent XSS attacks and work without modifications to the server. We have also demonstrated a testing framework that uses user-generated browsing sessions to measure the effects of security policy on the compatibility of web applications. Using our testing system we can examine web applications in detail to ensure that not only is the rendering not affected by security policy, but the application functionality remains intact.

8. ACKNOWLEDGMENTS

We would like to thank Anthony Cozzie for feedback on an early draft of this paper and we would like to thank Chris Munger for coming up with the Alhambra name. This research was funded in part by NSF grants CNS 0834738 and CNS 0831212, grant N0014-09-1-0743 from the Office of Naval Research, Intel and Microsoft under the Universal Parallel Computing Research Center, and AFOSR MURI grant FA9550-09-01-0539. Chris Grier was funded partially by NSF grants NSF-0433702 and NSF-0831535.

9. REFERENCES

[1] W3C Document Object Model. `http://www.w3.org/DOM/`.

[2] Alexa Internet, Inc. Alexa top 500 global sites. `http://www.alexa.com/topsites`.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, New York, NY, USA, 2008. ACM.

[5] P. Bisht and V. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 5th International Conference, Dimva 2008, Paris, France, July 10-11, 2008, Proceedings*, page 23. Springer, 2008.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault-Tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.

[8] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[9] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.

[10] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, 2007.

[11] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.

[12] D. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[13] G. Maone. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!, 2008. `http://noscript.net/`.

[14] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: Abstractions and software-hardware interface hardware-assisted deterministic multiprocessor replay. In *Proceedings of the 2009 Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009.

[15] Mozilla. Security/csp, 2009. `https://wiki.mozilla.org/Security/CSP`.

[16] Y. Nadji, P. Saxen, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In

Proceedings of the Network and Distributed System Security Symposium, February 2009.

[17] Perl.org. Perl taint mode. `http://perldoc.perl.org/perlsec.html`.

[18] D. Ross. IEBlog : IE8 Security Part IV: The XSS Filter, 2008. `http://blogs.msdn.com/ie/archive/2008/07/01/ie8-security-part-iv-the-xss-filter.aspx`.

[19] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, February 2009.

[20] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In *Proceedings of the 13th ACM conference on computer and communications security (CCS)*, 2006.

[21] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, June 2004.

[22] Symantec. Symantec Global Internet Security Threat Report Trends for July-December 07, April 2008. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[23] M. Ter Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings IEEE Symposium on Security and Privacy*, May 2009.

[24] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[25] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct 2007.

[26] WebKit. The webkit open source project. `http://www.webkit.org`.

[27] XSSed.com. XSSed - XSS (cross-site scripting) information and vulnerabile websites archive. `http://xssed.com`.

[28] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, August 2006.

[29] A. Yip, X. Wang, N. Zeldovich, and F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.