BrowserAudit: Automated Testing of Browser Security Features

Charlie
Hothersall-Thomas
Netcraft Ltd.
me@charlie.ht

Sergio Maffeis
Department of Computing
Imperial College London
maffeis@doc.ic.ac.uk

Chris Novakovic
Department of Computing
Imperial College London
c.novakovic@imperial.ac.uk

ABSTRACT

The security of the client-side of a web application relies on browser features such as cookies, the same-origin policy and HTTPS. As the client-side grows increasingly powerful and sophisticated, browser vendors have stepped up their offering of security mechanisms which can be leveraged to protect it. These are often introduced experimentally and informally and, as adoption increases, gradually become standardised (e.g., CSP, CORS and HSTS). Considering the diverse landscape of browser makes, releases, and customised versions for mobile and embedded devices, there is a compelling need for a systematic assessment of browser security.

We present BrowserAudit, a tool for testing that a deployed browser enforces the guarantees implied by the main standardised and experimental security mechanisms. It includes more than 400 fully-automated tests that exercise a broad range of security features, helping web users, application developers and security researchers making an informed security assessment of a deployed browser. We validate BrowserAudit by discovering both fresh and known security-related bugs in major browsers.

1. INTRODUCTION

Personal data, business transactions, critical infrastructure and even cars, refrigerators and lightbulbs are exposed through web interfaces to a wide variety of web browsers. Hence, the browser plays a key role in the modern information infrastructure, as the main gateway to access the information and capabilities made available online.

As such, browsers need to offer a variety of standardised security mechanisms which can be relied upon uniformly by the client-side of web applications, in order to deliver security guarantees to their users. For example, the Same Origin Policy (SOP) [17] is effective at preventing a range of cross-site scripting (XSS) attacks [21] against users' web browsers and is an integral aspect of modern web-based security. On the other hand, it is sometimes excessively strict; for instance, it forbids the sharing of information between different subdomains, a common requirement of large web sites. It is also coarse-grained, and several attempts have been made to enforce finer-grained access control [26, 23, 22] and origins [1, 7, 9, 13] in the browser. A variety of contemporary web browsers implement the Cross-Origin Resource Sharing (CORS) [34] standard, which may be used to control the flow of information between server-side resources and client-side scripts that attempt to access those resources via APIs. However, even fully-compliant implementations of the SOP and CORS mechanisms in some cases do not regulate access to other resources, such as images, embedded objects and web fonts, that can leave web applications vulnerable to CSRF [5, 11], clickjacking [19], framebusting [28] and CSS-based attacks [16]. The Content Security Policy (CSP) standard [31] enables much finer-grained control over the loading of arbitrary resources on a web page, mitigating several of these issues. These are just some examples of established and emerging security mechanisms offered by modern browsers.

Such mechanisms are often introduced experimentally and informally. As adoption increases, they gradually get standardised, and after numerous security reviews and bug reports they eventually can be relied on consistently across browsers [4, 20, 5]. Getting to that stage is not easy. For example, correctly implementing the CSP specification is non-trivial: it is a lengthy document with many cross-references to other standards and RFCs, many of which have been superseded by newer (and conflicting) standards and RFCs. It is possible that a browser vendor could incorrectly implement some part of the CSP and thus fail to provide some of its security guarantees to their users. There is therefore need for an automated tool that enables browser developers to complement low-level unit tests targeted to an individual code base with high-level testing of the effectiveness of the security features once the browser is deployed.

In this paper we introduce BrowserAudit, a framework for testing if a deployed browser correctly enforces the security guarantees implied by the main standardised security mechanisms. For practical purposes, we present BrowserAudit as a stand-alone web application that automatically tests the browser used to access it. BrowserAudit has been designed with different sets of users in mind. A casual web user can run the tests to gain a simple security assessment of their browser — critical, warnings or okay. With the recent surge of security breaches reported in the news, people are becoming increasingly security conscious and we believe there is a latent demand for tools that inform the public about security. A security researcher can benefit even more, viewing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '15
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

a detailed breakdown of each test result, and seeing which security features passed our tests and which had problems. We display textual descriptions for each category of tests and the source code of the tests. Browser developers can use BrowserAudit for debugging their security features and web developers can use it as a way to ascertain the security capability of users' browsers (Section 2). We chose to implement a careful selection of tests that covers both the most important browser security mechanisms (that should be implemented in any browser) and some of the most promising experimental ones that are not yet implemented by all of today's major browsers. Such browser security mechanisms aim to offer a number of security guarantees, allowing certain behaviours and forbidding others. BrowserAudit automatically tests over 400 behaviours where a certain action should either be allowed or blocked according to an implied browser security policy (Section 3).

We designed BrowserAudit to be efficient and scalable, and we evaluated its performance and its accuracy extensively by running it on a number of browsers and architectures. Using BrowserAudit, we have discovered several previously unknown security bugs in recent versions of Mozilla Firefox, which we have reported to the developers (Section 4.4).

While there are well-understood methodologies to generate unit tests for a given code base, there is no general solution to the problem of testing the end-to-end security behaviour of a family of applications (in our case web browsers) that must respect precise interoperability constraints (web standards) but can widely differ in implementation architectures, languages and design. Hence, we faced a significant challenge in order to develop our tests, carrying out a substantial amount of practical experimentation, guided by the official RFCs, our formal and informal models of web security, and a substantial body of academic and practical research on browser and web security (surveyed in Section 5.1). We believe that a fundamental contribution of BrowserAudit is to bring together in a single test suite a lot of explicit and implicit knowledge of the guarantees afforded by modern browser security mechanisms.

Although we believe that BrowserAudit is unique in its focus and breadth, we were inspired by a number of related web applications described in Section 5.2.

2. DESIGN OVERVIEW

The goals underlying the design of BrowserAudit are the following:

- *Wide coverage*: BrowserAudit should demonstrate that a wide range of browser security mechanisms can be tested automatically, reliably and efficiently. Complete test coverage of any such mechanism is not practically feasible, and beyond the scope of this project.[1]

- *Extensibility*: By its very nature, BrowserAudit will always be *work-in-progress*. As the browser threat landscape evolves, more tests will be needed to cover new security mechanisms, or to extend the coverage of existing ones. Our design should ease the task of creating, debugging and integrating additional test cases.

- *Ease of use*: BrowserAudit should be easily accessible on any modern browser connected to the Internet, without the need to install additional software. It should require no interaction from the user, otherwise running hundreds of tests would be impractical. Moreover, relying on user interaction would prevent the desired aim of running the tests transparently in the background.

- *Broad audience*: Our design should support a diverse range of users. A report on the security effectively offered by a deployed browser can benefit both browser developers, penetration testers, security researchers and typical web users.

- *Scalability*: Our design should be scalable on the server side. Several users may be testing their browser at the same time, and many security tests concern features that involve communicating with the server.

We now sketch the architecture of BrowserAudit and highlight the main design choices. We defer further implementation details to Sections 3 and 4.1.

2.1 User Experience

BrowserAudit is accessible by simply pointing the browser to be tested to `https://browseraudit.com/`. This is a landing page that briefly describes the aims of the project and contains a "Test me" button to move the user to the actual test page, hosted at `https://browseraudit.com/test`. This intermediate step avoids surprising users by actively requiring their consent to begin the testing phase. Once the user clicks to start the tests, the main testing loop initiates.[2]

BrowserAudit is completely automatic, and the user does not need to interact with the browser while it is being tested. As the tests are running, the user can see a progress bar advancing, and 4 test counters being incremented, as shown in Figure 1. For the benefit of typical web users, test runs are categorised using a simple Okay/Warning/Critical/Skipped traffic-light indicator. Okay denotes passed tests, Warning and Critical denote failed tests, and Skipped denotes tests that are skipped because the feature being tested is not supported by the browser. Failures regarding SOP, cookies, and the `Referer` header, which we consider the most crucial security features, are reported as Critical; failures regarding CSP, CORS, HSTS and the `X-Frame-Options` header are reported

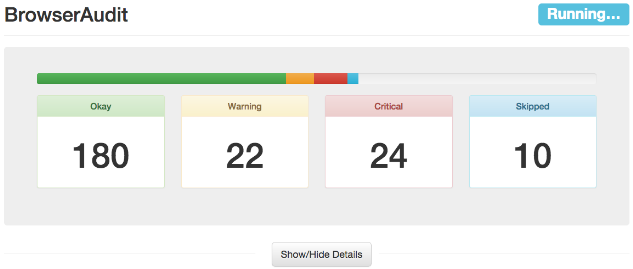

Figure 1: The test summary box part-way through the execution of our tests.

[1] For example, an exhaustive test of the Same Origin Policy would also need to demonstrate that, for any domains A and B, a page from domain A cannot access certain properties of a page from an incompatible domain B.

[2] Unless JavaScript is disabled, in which case we display a warning to the user. Automated tests cannot be run without JavaScript, and some security features need JavaScript in order to be exercised.

as Warnings. This distinction is somewhat arbitrary, and will change as such features become more broadly supported and new ones are introduced.

After the test suite has finished running, the grey background of the summary box takes the colour of the worst failed test, if any, or becomes green if all tests passed. This traffic-light indicator provides a basic level of information about the current level of security offered by the browser.

More sophisticated users, such as security researchers or browser developers, need more information on the tests performed and on their outcomes. Clicking on the "Show/Hide Details" button displays a summary box that shows the various categories of tests (reflecting the security mechanisms that have been tested), and the number of failed tests for each of them, as shown in Figure 2.

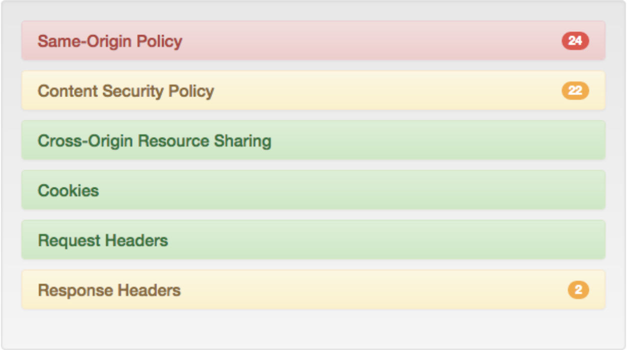
Same-Origin Policy 24
Content Security Policy 22
Cross-Origin Resource Sharing
Cookies
Request Headers
Response Headers 2

Figure 2: BrowserAudit summary box.

Each category can be expanded and collapsed to show a description of the corresponding security mechanism, and a list of sub-headers that in turn can be expanded to reveal individual tests for a specific feature, as illustrated in Figure 3. For each individual test we show a descriptive title that can be clicked to show the source code of the test itself. The look of our design is based on Twitter's front-end framework Bootstrap[3], which makes it easy to produce a layout that works consistently across browsers and devices.

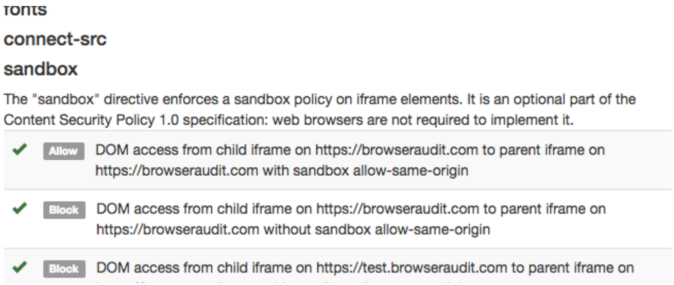
fonts
connect-src
sandbox
The "sandbox" directive enforces a sandbox policy on iframe elements. It is an optional part of the Content Security Policy 1.0 specification: web browsers are not required to implement it.
✔ Allow DOM access from child iframe on https://browseraudit.com to parent iframe on https://browseraudit.com with sandbox allow-same-origin
✔ Block DOM access from child iframe on https://browseraudit.com to parent iframe on https://browseraudit.com without sandbox allow-same-origin
✔ Block DOM access from child iframe on https://test.browseraudit.com to parent iframe on

Figure 3: Some sub-categories of CSP tests, with expandable test titles and result indicators.

2.2 Architecture

The client side and server side of BrowserAudit work together in order to run tests in the browser. Put simply, the server side exercises browser security features, and the client side tests that these features are implemented as expected.

When multiple concurrent users access BrowserAudit, we need to avoid congestion on the server side, as testing each browser causes a bursty interaction with the BrowserAudit server, in the form of hundreds of requests per user per minute. For this reason, we adopt the architecture illustrated in Figure 4, consisting of two web servers: a public-facing Nginx web server and a Go[4] application server. The Nginx server is running as a *reverse proxy* in front of the Go server, which is not publicly accessible.

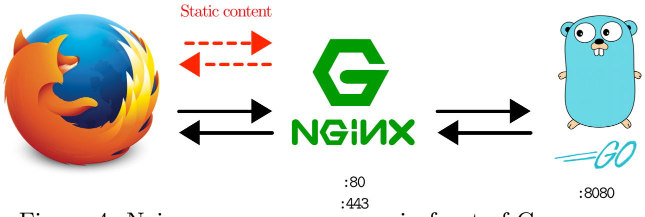

Figure 4: Nginx as a reverse proxy in front of Go server.

When the Nginx server receives HTTP requests for static resources, such as our JavaScript tests, it responds by directly fetching the resource from the local `static/` directory. Dynamic requests are instead proxied to the Go server, and the Go responses are forwarded back to the client. Nginx also handles SSL termination, caching, gzip compression, URL rewriting, and keeps access and error logs. This architecture reduces the load on the Go server, which can focus on serving only dynamic requests that depend on the user's session. We chose Go for its high performance, its support for concurrency (which we leverage indirectly via the `net/http` package) and its ease of use.

This architecture also has the security benefit that our Go application can run as a non-privileged user, since it only needs to bind to port 8080 and not the privileged ports 80 and 443, which are bound by Nginx instead.

Certificates. In order to ensure a good coverage of various security features that involve the use of Strict Transport Security and cross-origin testing, BrowserAudit makes use of four domains: `browseraudit.com`, `test.browseraudit.com`, `browseraudit.org` and `test.browseraudit.org`. The server holds a single SSL certificate that is valid for both.

Sessions. We use sessions to keep track of intermediate test results and other test-related data for each user whilst their tests are in progress. Sessions are needed because in many of our security tests, it is the *server* that makes the decision as to whether or not the browser passed the test, not the test framework running in the browser. In these cases, the client must send an additional request asking the server what the test result was, so that it can be displayed to the user.

Caching. In our tests, there are many cases in which a request is first made to store a default result on the server, and then a second request *may* be sent to overwrite this result, depending on whether or not the browser correctly implements a given security feature. If a user runs the tests twice in short succession, and this second result was cached and therefore did not reach our server, our application would report an incorrect test result. We ensure that this cannot happen by preventing HTTP responses from being cached.

2.3 Tests

A typical test of a security feature involves making multiple Ajax or image requests to the server and checking if the actual responses match the expected responses.

[3] `http://getbootstrap.com/`

[4] `https://golang.org/`

JavaScript and libraries. Our tests are written directly in JavaScript, for convenience augmented with the jQuery library.[5] We deploy our tests using the Mocha[6] framework for browser-based JavaScript unit testing, with some custom modifications to improve the output layout. Mocha supports asynchronous tests, and handles the test automation and the display of results to the user, so that the effort of test design can be concentrated on the security feature to be tested. Tests can be categorised using the `describe()` function. Multiple `describe()`s can be placed inside each other, resulting in hierarchical categories in Mocha's output.

```
describe("HTTP Response Headers", function() {
  describe("Strict-Transport-Security", function() {
    it("HSTS should expire after max-age", function(done){/* actual test ... */});
  });
});
```

Figure 5: Writing a test inside categories with Mocha.

The code in Figure 5 shows a section for HTTP Strict Transport Security (HSTS), which is nested beneath an HTTP Response Headers category. Inside the HSTS category, there is a single test, denoted by the call to the `it()` function, whose first parameter is a test description and second parameter is a function implementing the test proper.

```
1 $.get("/del_httponly_cookie", function() {
2   expect($.cookie("httpOnlyCookie")).to.be.undefined;
3   $.get("/set_httponly_cookie", function() {
4     expect($.cookie("httpOnlyCookie")).to.be.undefined;
5     done();
6   });
7 });
```

Figure 6: The client side of a proof-of-concept `HttpOnly` cookie test.

Figure 6 shows a proof-of-concept test to check that the browser correctly implements `HttpOnly` cookies (see Section 3.4). Line 1 loads a page to clear any leftover cookies from previous test runs, line 2 checks that the cookie is not defined, line 3 loads a second page that sets the cookie, and line 4 checks that we are unable to read it. The call to `done()` on line 5 informs Mocha that the asynchronous test is complete. In order to make the source code of the tests easier to understand and maintain, we are also leveraging the Chai[7] assertion library.

Tests. In most cases, we automatically generate the JavaScript code of tests that have a similar structure, but depend on different parameters. For example, in Figure 7 we show the most interesting parts of the `ajaxSopTest` function, which generates Mocha code for testing AJAX calls with respect to the SOP. The choice of the right parameters for the resources to load (`defaultResults`, `iframeSrc`) are crucial to the correctness of each test instance. To favour modularity and coverage, we instantiate a separate Mocha test for each case to be tested, rather than bundling a large number of cases in the same test. To ensure maximum portability, we implement as much as possible on the client side using standard, browser-independent features.

```
function ajaxSopTest(globalTestId, shouldBeBlocked,sourcePrefix, destPrefix) {

  // omitted code:  variable initialisation

  var test_template = function(done) {
    $.get("/sop/"+defaultResult+"/"+id,
      function() {$("<iframe>", { src: iframeSrc })
        .css("visibility", "hidden")
        .appendTo("body").load(function() {
          $.get("/sop/result/"+id,function(result) { expect(result).to.equal("pass"); done();});});});};

  // omitted code:  save source code for display

  browserAuditTest(globalTestId, test_template);
}
```

Figure 7: Code to generate SOP tests for AJAX calls.

Whenever possible, we write asynchronous tests using callback patterns rather than timeouts. We annotate the titles of tests whose results depend on timeouts with a small clock icon. We try to avoid using timeouts because, when a timeout expires, it is not possible to distinguish a true test failure from an anomalous delay in a browser event or network connection. Moreover, it is difficult to estimate appropriate timeout values for many events. For certain tests, however, we cannot avoid using timeouts. For example, to detect whether a CSP policy that denies the use of JavaScript but allows the loading of fonts in an iframe is enforced correctly, the BrowserAudit test framework needs to give time for the iframe to try to load the font, and then ask the server if the font was requested. We are not allowed to run JavaScript in the iframe to inspect the page and detect if the font was loaded; likewise, we cannot ask the user for confirmation, because our tests must run without user interaction.

3. BROWSER SECURITY MECHANISMS

In this section, we describe the range of security mechanisms currently exercised by BrowserAudit. Each mechanism induces, sometimes implicitly, a security policy. Our emphasis is on testing representative instances of behaviours that should be allowed or blocked according to the corresponding security policy.

3.1 Same-Origin Policy

In the early days of the web, there was little incentive to control the resources that could be included in a web page: most web pages were static, and web developers were free to include resources (e.g., images) from any source in their web pages. As web sites became dynamic and interactive, thus allowing web developers to include user-supplied content in their pages, and requiring web browsers to execute scripts supplied by the web server, browser vendors became more security-aware: they recognised that permitting the execution of arbitrary code (e.g., JavaScript) from untrustworthy sources was potentially dangerous, and began to impose restrictions on the execution of scripts from "foreign" locations.

[5] http://jquery.com/
[6] http://visionmedia.github.io/mocha/
[7] http://chaijs.com/

In particular, “foreign” scripts were forbidden from accessing the Document Object Model (DOM) — the browser’s internal hierarchical representation — of the web page in which the script was included. These are the foundations of the *same-origin policy* (SOP) [17], still implemented in contemporary web browsers: a script is only permitted to access the DOM of a loaded web page if their schemes, hostnames and port numbers in their URIs — their *origins* — match.

There are mechanisms for relaxing the SOP so that information can be shared between DOMs with differing origins; the easiest method of doing so is to set the same `document.domain` property in each DOM, so that the web browser considers the DOMs to have the same origin.

BrowserAudit comprehensively exercises a web browser’s implementation of the SOP and the mechanisms for relaxing it to ensure that inter-DOM access is permitted when both DOMs are deemed to have the same origin, and forbidden at all other times. The basic structure of our SOP tests for the DOM follows a standard pattern: scripts running on web pages loaded in nested iframes manipulate the DOM’s `document.domain` property, and the script from one iframe attempts to access the DOM of the other iframe. Each test exercises a particular combination of the following parameters:

- the domain from which the web page loaded by the parent iframe is served (one of `browseraudit.{com/org}` or `test.browseraudit.{com/org}`);
- the domain from which the web page loaded by the child iframe is served (also selected from the list above, and potentially the same domain used by the parent iframe’s web page);
- the value of `document.domain` to be set by a script running in the parent iframe;
- the value of `document.domain` to be set by a script running in the child iframe; and
- the direction in which the DOM access is attempted (parent iframe to child, or child iframe to parent).

The client-side test framework checks whether the web browser satisfies the SOP by selecting combinations of these parameters that should be allowed or blocked by the SOP and verifying that the correct behaviour is observed.

For example, Figure 8 shows a diagram for a test in which a parent iframe tries to access the DOM of its child iframe. The parent is loaded from `https://browseraudit.org` whereas the child is loaded from `https://test.browseraudit.org`. We expect this access to be blocked since we are not setting any `document.domain` values in this test, and the hosts are not the same. Note how we load images from specially-crafted addresses (such as `https://browseraudit.com/sop/[pass/fail]/TEST_ID`) to communicate test results from the server, thus avoiding restrictions imposed by the SOP itself.

In general, if a script running in either iframe is able to access the DOM of the other, the script notifies the BrowserAudit server that access to the other iframe’s DOM was granted; the test framework then queries the server for whether this notification was sent. If the notification was sent and DOM access was expected given the chosen test parameters, or if the notification was *not* sent and DOM access was *not* expected given the chosen test parameters,

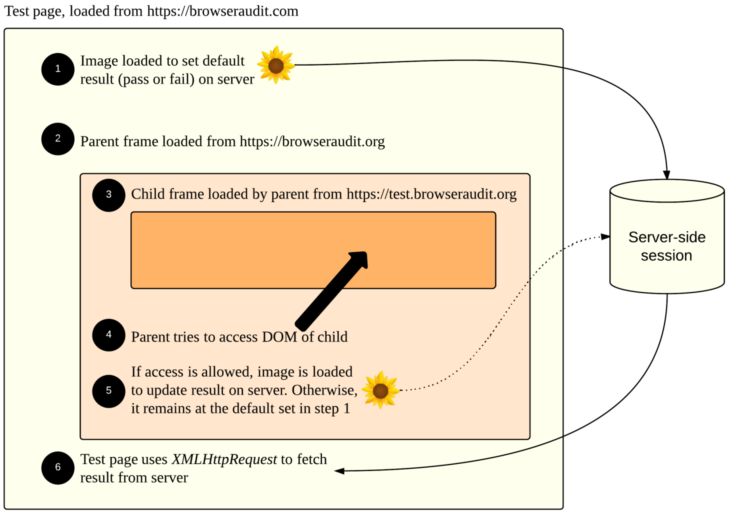
Test page, loaded from https://browseraudit.com
1
Image loaded to set default result (pass or fail) on server
2
Parent frame loaded from https://browseraudit.org
3
Child frame loaded by parent from https://test.browseraudit.org
4
Parent tries to access DOM of child
5
If access is allowed, image is loaded to update result on server. Otherwise, it remains at the default set in step 1
6
Test page uses *XMLHttpRequest* to fetch result from server
Server-side session

Figure 8: An example of a SOP test in which the parent frame tries to access the DOM of its child.

the test framework considers the browser to have passed that particular test; otherwise, the browser permitted insecure DOM access and is considered to have failed the test.

The SOP applies not only to DOM access, but also to cookies with differing paths and HTTP requests made to other domains via the XMLHttpRequest API; BrowserAudit also tests a browser’s implementation of the SOP for all of these features, providing a total of 84 SOP tests, generated by 4 JavaScript templates.

3.2 Cross-Origin Resource Sharing

Cross-origin resource sharing (CORS) [34] is a flexible standard for relaxing the SOP that selectively permits resources to be shared across origins; it is implemented in APIs capable of initiating cross-origin resource requests (e.g., XMLHttpRequest) in a range of modern web browsers. It allows a client to include a resource from a server with a different origin only if the resource request is explicitly authorised by the server. This is achieved via two additional HTTP headers: an `Origin` header is sent by the client as part of the request and specifies the origin of the resource attempting to use the cross-origin resource, and an `Access-Control-Allow-Origin` header is sent by the server as part of the response and specifies the origins from which this resource may be used, effectively ordering the client to uphold or relax the SOP for this resource request.

The majority of cross-origin requests made using CORS are “simple”, defined in the CORS specification [34] as an HTTP request with one of `GET`, `POST` or `HEAD` as the request method and headers from a narrowly-defined whitelist (`Accept`, language-related headers and a small number of acceptable `Content-Types`). Other requests are deemed “non-simple”; the CORS specification requires that the client precedes such requests with a “preflight” request that includes further detail so that the server can more accurately decide whether or not to allow the cross-origin request (although, in reality, some browsers misclassify simple and non-simple requests). In response to the preflight request, the server sends additional headers: `Access-Control-Allow-`

Methods, a comma-delimited list of HTTP methods permitted to be used to access the resource; Access-Control-Allow-Headers, a comma-delimited list of headers that may be sent with the main CORS request; and Access-Control-Expose-Headers, a list of headers that should be exposed to the requester (e.g., a script accessing a resource using XMLHttpRequest). If the main CORS request violates either of the restrictions imposed by the Access-Control-Allow headers, the main request is considered a violation of the SOP and is aborted.

BrowserAudit tests the intended behaviour of CORS by sending a series of cross-origin XMLHttpRequest requests from the browser and verifying that the client exhibits CORS-compliant behaviour when the BrowserAudit server sends a response containing a range of CORS HTTP headers. The testing methodology is similar to that for the SOP, described in Section 3.1: the client attempts to retrieve a file from the BrowserAudit server, and sends a notification to the BrowserAudit server if this retrieval was successful. The BrowserAudit test framework then queries the server for whether the notification was sent. If the notification was sent for CORS-compliant requests and *not* sent for CORS-violating requests, the browser is deemed to correctly implement the CORS standard; if a notification was sent for CORS-violating requests, or if one was *not* sent for CORS-compliant requests, the browser is considered to lack full compliance.

We currently run 54 tests of different CORS scenarios, automatically generated by 4 JavaScript test templates.

3.3 Content Security Policy

The *Content Security Policy* (CSP) standard [31] enables much finer-grained control over the loading of arbitrary resources on a web page that what is allowed by the SOP and by CORS.[8] As with CORS, a content security policy is delivered via an HTTP header (or via a <meta> element in the HTML header); the CSP specification states that the Content-Security-Policy header should be used for this purpose.

The header allows servers to declare to CSP-compliant clients the permitted origins of a range of resources: images, stylesheets, scripts, web fonts, embedded objects and other types of resource may all be controlled by a single policy. *Directives* may be used to restrict the origins of these different types of resource independently of each other, and a "default" directive may be used to restrict the origins of all resources that are not explicitly controlled elsewhere in the policy. For example, a server at example.com serving web pages to CSP-compliant browsers could restrict the loading of images to those hosted on the same server and the loading of embedded objects (such as Java applets) to those hosted on a trusted server at applets.example.com (and thus forbid embedded objects and images from being loaded from other origins) by specifying the following value for the Content-Security-Policy HTTP header:

```
image-src 'self'; object-src http://applets.example.com
```

When served alongside a web page to a CSP-compliant web browser, such policies can preempt many of the attacks described at the beginning of this section; e.g., using the script-src directive to control the permissible origins of scripts mitigates the effects of CSRF, clickjacking and frame-busting (since they rely primarily on successful JavaScript injection), and using the style-src directive to control the permissible origins of stylesheets defeats CSS-based attacks. Note that one cannot specify which specific resources may be loaded from these other origins: permitting a particular Java applet to be loaded from applets.example.com also permits *any other* embeddable object to be loaded from applets.example.com, so whitelisted origins should be trustworthy (particularly those granting the power to execute arbitrary code, such as script-src).

The CSP standard also includes a mechanism for reporting violations of a given policy via a special report-uri directive; this directive defines a URL to which a *violation report* should be sent.

BrowserAudit exercises a browser's CSP implementation by performing a battery of tests on each directive defined in the CSP specification, as well as the violation-reporting capabilities of the report-uri directive. Similarly to the SOP tests (described in Section 3.1), each CSP test attempts to load a resource inside an iframe using a particular combination of the following parameters:

- the domain from which the web page loaded by the iframe is served (one of browseraudit.com or test.browseraudit.com);
- the domain from which the desired resource is requested (also selected from the list above, and potentially the same domain used by the iframe's web page); and
- the CSP imposed on the iframe by the BrowserAudit server via the Content-Security-Policy header.

We run around 220 CSP tests, mostly generated by three JavaScript templates, that in turn load approximately 280 iframes representing particular behaviours to be tested. In each test, the browser is expected to either allow or block access to the given resource, and the act of requesting the resource from the BrowserAudit server allows it to track violations of the given CSP. On the client side, the BrowserAudit test framework queries the server after the iframe has loaded to find whether the browser accessed the resource and therefore determine whether the browser exhibited the behaviour expected of a CSP-compliant browser: allowing a request permitted by the given CSP or blocking a request restricted by the CSP is regarded as a correct implementation of the standard and thus a test success, while an attempt to access the resource when given a restrictive CSP or a failure to request the resource when given a permissive CSP is regarded as an erroneous implementation of the standard and thus a test failure.

Figure 9 shows the test code for one of our CSP tests. The code runs on the main BrowserAudit page and loads an outer iframe from browseraudit.com with the CSP header sandbox allow-same-origin allow-scripts. This outer iframe is very simple (Figure 10), and its role is just to load an inner iframe from browseraudit.com that is subject to the given policy: scripts can run, and have same-origin permissions.

The inner frame, whose code is shown in Figure 11, tries to perform an XMLHttpRequest to test.browseraudit.com, which should be blocked. Note that since we cannot rely on

[8] We concern ourselves only with version 1.0 of the Content Security Policy standard, as its successor (version 1.1) is still in Working Draft status at the time of writing; however, the two versions are similar, and the latter can be viewed as an extension of the former.

```
$("<iframe>", { src: "/csp/serve/206/param-html?policy=‘sandbox allow-same-origin allow-scripts’&defaultResult=pass" })
  .css("visibility", "hidden").appendTo("body").load(function() {
    $.get("/csp/result/206", function(result) {
      expect(result).to.equal("pass");
      done();
        });
  });
```

Figure 9: A CSP test exercising the browser's implementation of the `sandbox` directive.

```
<html><body>
  <iframe src="/csp/serve/206/param-htmlb?browseraudit=sessionCookie"></iframe>
</body></html>
```

Figure 10: The HTML for the outer iframe loaded by the test script shown in Figure 9.

user credentials to be sent with synchronous XMLHttpRequests, we pass the session cookie (abstracted for readability in Figure 11 as `sessionCookie`) as a parameter of the request. All of this information is also visible to the BrowserAudit user by clicking on the corresponding test title in the user interface.

```
<html><body>
  <script>
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "https://test.browseraudit.com/csp/serve/206/oktext?browseraudit=sessionCookie&corsOrigin=browseraudit.com&corsMethod=GET", false);
    xhr.send(null);
    if (xhr.status == 200) {
      var img = document.createElement("img");
      img.setAttribute("src", "/csp/fail/206/png");
      document.body.appendChild(img);}
  </script>
</body></html>
```

Figure 11: The HTML for the inner iframe corresponding to the outer iframe shown in Figure 10.

3.4 Cookies

In our tests for the SOP (Section 3.1) we explore the security implications of setting the cookie scope through the `Domain` and `Path` attributes. There are two other important aspects of cookie security: the `HttpOnly` and `Secure` attributes. We test the behaviour of these attributes as defined in RFC 6265, "HTTP state management mechanism" [2].

The `HttpOnly` attribute of a cookie instructs the browser to reveal that cookie only through an HTTP request; i.e., it should not be made available to client-side scripts. The benefit of this is that, even if a cross-site scripting (XSS) vulnerability is exploited, the cookie cannot be stolen. `HttpOnly` cookies are supported by all major browsers, with the notable exception of Android 2.3's stock browser. BrowserAudit includes tests that check that an `HttpOnly` cookie sent from the server cannot then be accessed by JavaScript, and that `HttpOnly` cookies cannot be created by JavaScript.

When a cookie has the `Secure` attribute set, a compliant browser will include the cookie in an HTTP request only if the request is transmitted over a secure channel, i.e. an HTTPS request. This keeps the cookie confidential: an attacker would not be able to read it even if he were able to intercept the connection between the victim and the destination server. The `Secure` attribute is supported by all major browsers. BrowserAudit includes tests checking the behaviour of the `Secure` attribute both when the cookies are set by the server and set by JavaScript.

3.5 Referer Header

The `Referer` header should not be included in a non-secure request if the referring page was transferred with a secure protocol. This behaviour is defined in RFC 2616, "Hypertext transfer protocol HTTP/1.1" [15]. This behaviour exists because the referrer might disclose an otherwise private information source. In BrowserAudit, we test this behaviour by loading a web page over HTTPS containing an image loaded over HTTP and checking that the `Referer` header was not sent to the server with request for the image.

3.6 Response Headers

3.6.1 X-Frame-Options

`X-Frame-Options` is a server-side technique that can be used to prevent clickjacking (UI redressing) attacks. Its implementation in current browsers is documented in RFC 7034, "HTTP Header Field `X-Frame-Options`" [27]. `X-Frame-Options` is a response header that specifies whether or not the document being served is allowed to be rendered in a frame; more specifically, the header specifies the origin (scheme, host and port) that is allowed to render the document in a frame. BrowserAudit tests for correct behaviour of the `DENY`, `SAMEORIGIN` and `ALLOW-FROM` header values. Our tests use only `<iframe>` elements, although the header can also apply to `<frame>`, `<object>`, `<applet>` and `<embed>` elements.

`X-Frame-Options` is supported in all modern browsers, although the implementations across browsers differ. Some browsers behave differently when dealing with nested frames, so we do not test these cases as there is no defined correct behaviour. Note also that not all browsers support the `ALLOW-FROM` value.

3.6.2 Strict-Transport-Security

HTTP Strict Transport Security (HSTS) is a security mechanism that allows a server to instruct browsers only to communicate with it over a secure (HTTPS) connection for that domain. It exists primarily to defend against man-in-the-middle attacks in which an attacker is able to intercept his victim's network connection [20]. The server sends this instruction with a `Strict-Transport-Security` header, as defined in RFC 6797, "HTTP Strict Transport Security (HSTS)" [18].

When HSTS is enabled on a domain, a compliant browser must rewrite any plain HTTP requests to that domain to use HTTPS. This includes both URLs entered into the navigation bar by the user, and resources included on a web page. The `Strict-Transport-Security` header should only be sent in an HTTPS response. If the browser receives the header in a response sent over plain HTTP, it should be ignored.

In BrowserAudit, we test the basic behaviour of HSTS and its `includeSubDomains` directive. We also ensure that the header is ignored when transferred via an insecure protocol, and that the HSTS state correctly expires based on the `max-age` value set in the header. All of these tests work by testing whether a request for an image at `http://browseraudit.com/set_protocol` is rewritten to use HTTPS or not.

Several current browsers support HSTS, with the notable exception of Internet Explorer, which does not support it even in the latest available version (11). Safari has only supported HSTS since version 7.

4. EVALUATION

4.1 Performance

A primary concern of BrowserAudit is scalability, given that a single invocation of the full test suite involves approximately 1,500 requests and around 3MB of data being transferred between the browser and server. The server must handle all of these requests quickly (ideally in under 300ms), given the large number of tests in the BrowserAudit test suite and the reliance of some of the tests on timeouts (see Section 2.3).

The BrowserAudit web and database servers are currently hosted on a single virtualised server with two CPU cores and 2GB of memory, running Ubuntu 14.04. We evaluated BrowserAudit's server-side performance by running the BrowserAudit test suite in 15 web browsers repeatedly and concurrently for 15 minutes. Over this period, the BrowserAudit server handled around 225,000 requests and served a total of 450MB of data. The 1- and 5-minute load averages on the BrowserAudit server are shown in Figure 12; the peak load averages over the 15-minute duration of the performance test are 1.2 and 0.7 respectively, where a load average of 1 indicates that a single CPU core is operating at capacity. Based on these performance figures, we estimate that a single BrowserAudit application server using this configuration could comfortably support up to 25 concurrent test suite executions.

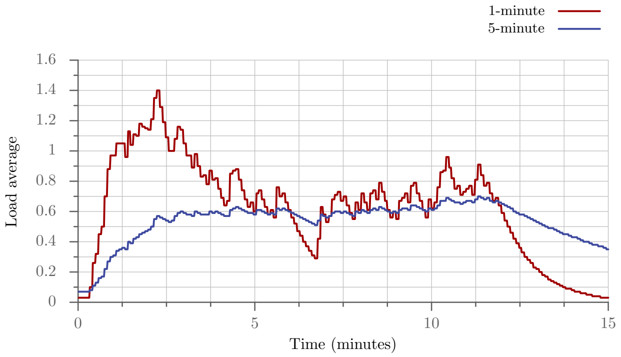

Figure 12: The 1- and 5-minute load averages on the BrowserAudit server during the performance evaluation.

As described in Section 2.2, our design is ready to be scaled up as the BrowserAudit user base grows. Nginx can be configured as a load balancer, passing requests to one of many application servers. Deploying Go application server instances is trivial thanks to Go's ability to compile a program to a single statically-linked binary, so there is no dependency chain. In order to maintain session persistence, Nginx's `ip_hash` directive can be used to ensure that all requests from the same IP reach the same application server, maintaining the integrity of a single suite execution.

Most client-side tests contain components that are loaded synchronously inside dynamically-created `<iframe>` elements, which become redundant as soon as the test result is reported in the browser. Over time, the DOM of the main BrowserAudit window would amass an overwhelming number of iframes, slowing down the execution of tests as the browser struggles to create and append additional iframes. We avoid this problem by dynamically removing any iframes appended to the DOM during each test's tear-down phase (via Mocha's `afterEach()` routine). We run 15 repetitions of 10 concurrent executions of the whole test suite on a 64-bit Windows 7 machine with a 6-core Intel i7 4930K CPU and 64GB of memory, and Chromium 40.0.2205.0. Under these conditions, the average execution time for a test run on the client side is just over a minute. By contrast, a single run in Safari 600.1.4 on an iPhone 5 with iOS 8.1 takes on average 1.35 minutes, skipping 24 tests. Overall, the client execution time varies quite broadly across browsers and architectures, but considering these figures we find it an acceptable cost for a browser security scan.

4.2 Correctness

Verifying the correctness of our tests is challenging, as they need to convey in a final pass or fail result a whole security-sensitive behaviour: a test containing a small bug could still pass, which is generally the expected result for browsers correctly implementing a given security mechanism.

Of course, no web browsers contain intentional security flaws that would allow us to verify the correctness of tests. Modifying the source code of existing open-source browsers to break their security features in order to ensure that tests fail when expected is possible but challenging given the complexity of modern web browser codebases.

However, it is a matter of public record that some web browsers either do not implement some of the security mechanisms tested by BrowserAudit, or only implement subsets of those security mechanisms. We leverage the results of browser-profiling projects such as *Browserscope*[9] and *CanIUse*[10] to verify whether BrowserAudit tests are correctly implemented; e.g., tests that verify blocking behaviour should pass in browsers known to correctly implement all aspects of a security mechanism and fail in browsers that do not. We made use of this workflow extensively during the development of BrowserAudit tests to ensure that they behave as expected.

Using *BrowserStack*,[11] a web-based browser testing service, we have evaluated BrowserAudit on a range of browsers on a number of different operating systems, across both desktop and mobile platforms. The full BrowserAudit test suite runs reliably in Safari 6, Firefox 13 and Chrome 25 or more recent versions, automatically skipping tests where a feature is not supported. BrowserAudit also runs correctly on Internet Explorer 11, but due to problems related to Mocha and the limited call stack of IE, we cannot run the whole test suite in one go. On older versions of these browsers we can run selected subsets of the BrowserAudit tests.

4.3 Test coverage

We have noted in Section 2 that a full coverage of browser security features is unattainable. Here we discuss a number

[9] `http://www.browserscope.org`

[10] `http://caniuse.com`

[11] `http://www.browserstack.com`

of security features not covered by BrowserAudit, but that we believe can be added to our framework.

We imply in Section 3 that there is no single same-origin policy but rather a collection of related security mechanisms. We currently test the same-origin policy for DOM access, XMLHttpRequest and cookies. This could be expanded on to test the same-origin policies for Flash, Java, Silverlight, and HTML5 web storage.

The postMessage API is used by many developers to avoid the headaches sometimes inflicted by the same-origin policy [4]. Since the API allows the sender of a message to specify the origins of the recipients that may receive the message, there are lots of origin-related tests that we could write for this feature in BrowserAudit.

Another security feature that could be eventually covered by BrowserAudit is the `X-Content-Type-Options` response header first introduced in Internet Explorer 8.[12] It is now also supported by Chromium and Safari, whilst the Firefox team is still debating its implementation.[13] It is designed to prevent browser-sniffing attacks where a resource sent with an innocuous MIME-type, such as `text/plain`, but formatted as for example HTML, is erroneously rendered as an HTML resource, with the obvious security implications [3].

In Sections 3.1–3.4 we have discussed how to extend coverage of features for which we already have some tests. Summarising, the main limitations are that: in many tests involving origin mismatches, we only test origins that differ in host rather than scheme or port; we do not test CSP directives where a resource is loaded from a URL that redirects; we do not test that cookies cannot be set for top-level domains that include a country code, such as `co.uk` (whereas they should be settable for `example.uk`). At the moment we also do not have a test for the `Report-Only` header of the CSP, but again this was due to time constraints rather than a limitation of the BrowserAudit framework and could be addressed in future work.

Finally, cryptographic APIs such as the W3C WebCrypto API and the OpenSSL library are key pieces of the browser security puzzle, but cryptographic testing is beyond the scope of BrowserAudit and better left to dedicated projects such as *How's My SSL?*.[14]

4.4 Uncovering Security Bugs

BrowserAudit's tests have uncovered two previously-unknown bugs in Firefox's implementation of the CSP standard; these bugs are present in all versions of Firefox that implement the CSP standard up to version 32.0.3. The first bug[15] allows the loading of same-origin stylesheets with the policy

```
default-src ‘none’; style-src ‘unsafe-inline’;
```

similarly, the second bug[16] allows the loading of same-origin Worker and SharedWorker objects in scripts with the policy

```
default-src ‘none’; script-src ‘unsafe-inline’.
```

In both cases, the `‘unsafe-inline’` declaration in the policy states that only inline stylesheets and scripts must be permitted: external resources, even those from the same origin, must be blocked. We reported both of these bugs to Mozilla during the version 29 release cycle, and they were fixed in version 33 of Firefox.

Firefox does not currently implement the `sandbox` CSP directive; this optional feature of the CSP 1.0 specification directs browsers to relax the given security controls on iframes embedded in the page, as if they had been supplied in the `sandbox` attribute of each `<iframe>` element. The `sandbox` attribute is in fact a feature of the HTML5 specification [17] and states that an iframe containing a `sandbox` attribute should have *all* security controls enabled unless specifically whitelisted by values inside the `sandbox` attribute. Development work on the implementation of this directive in Firefox is currently underway.[17] However, the current implementation does not correctly handle the case where an empty value is given for the `sandbox` CSP directive; the CSP 1.0 specification implies that the browser should apply a `sandbox` attribute with an empty value (and thus enforce a highly-restrictive sandboxing policy — a view also taken by other browsers, such as Chromium), but Firefox's implementation does not apply a `sandbox` attribute at all in this scenario (thus failing to enforce *any* sandboxing policy). This flaw was uncovered by the current set of CSP tests in BrowserAudit, and we are in discussion with Firefox developers to address it before their `sandbox` implementation lands in a stable version of the browser.

5. RELATED WORK

In this section we discuss some related work on browser security, which influenced the design of our tests, and review some web applications that perform security-relevant tests, which served as a source of inspiration for BrowserAudit.

5.1 Browser Security

The authoritative sources of information on upcoming browser security mechanisms are of course the W3C RFCs and Drafts such as [17, 31, 6, 34, 18]. Most security measures are the result of a lot of practical experimentation and academic research that led to proposals that gradually gained adoption and became more robust through security reviews and public scrutiny. A paradigmatic example are the early contributions of Barth, Jackson *et al.* to postMessage, the `origin` header and HTTPS [4, 20, 5].

The standards themselves provide a lot of detail about the intended security behaviour, but additional research is needed to interpret the consequences for deployed web applications. For example, De Ryck *et al.* perform a security analysis of some of the upcoming standards in [12], finding them to be be of high quality but also highlighting potential security risks. Singh *et al.* [30] discover potentially dangerous incoherencies amongst different browser access control policies.

A broad, in-depth analysis of browser security can be found is Zalewski's online *Browser Security Handbook* [36], and the companion book *The Tangled Web* [37]. These resources gather a wealth of information on browser security features, their shortcomings and the peculiar differences in browser support.

[12] `http://blogs.msdn.com/b/ie/archive/2008/09/02/ie8-security-part-vi-beta-2-update.aspx`

[13] `https://bugzilla.mozilla.org/show_bug.cgi?id=471020`

[14] `https://www.howsmyssl.com`

[15] `https://bugzilla.mozilla.org/show_bug.cgi?id=1007205`

[16] `https://bugzilla.mozilla.org/show_bug.cgi?id=1007634`

[17] `https://bugzilla.mozilla.org/show_bug.cgi?id=671389`

Other efforts that have variously influenced our work, and the applications discussed in the next section, have focussed on large-scale security analyses [33, 10, 24], empirical studies [8, 29, 35], user tracking and fingerprinting [25, 32, 25].

5.2 Web Sites

Panopticlick[18] is an experiment to investigate how unique — and therefore trackable — modern web browsers are, by fingerprinting their version and configuration information. Some of this information can be gleaned directly from browser requests, whereas other information is made available by the presence of JavaScript and browser plugins. Visitors click a "Test Me" button and are then provided with their browser's uniqueness score and a breakdown of the measurements used to obtain the result. These data are then anonymously stored in the project database to make future uniqueness scores more accurate, and to allow for analysis of the data, as discussed in [14]. Although focussed on privacy rather than security, *Panopticlick* was the main inspiration for our project.

BrowserSpy[19] is another web site that reports how much information can be retrieved from a browser by visiting a test page. Its focus is on privacy, yet some of its tests are security-related, although not presented as such. For example, one test checks that JavaScript cannot read `HttpOnly` cookies. Each of *BrowserSpy*'s current 75 tests has to be run individually, since the output is rather verbose, and the output does not show implementation details that could be useful for a technical audience. In contrast, our 400+ tests run automatically, and advanced users can see the actual code of each individual test.

How's My SSL?[20] is a recent project that advises the user on the security of their TLS client (web browsers act as TLS clients when engaged in HTTPS communication). It works by running a TLS server that has been modified so that the client-server handshake is exposed to the web application, allowing it to inspect the cipher suites that the client supports and perform a security assessment. The results are reported clearly, with "Learn More" links for more technical background which also inspired our design. The test results can be accessed via a JSON API, and could be potentially integrated on BrowserAudit to complement our tests. Qualys' *SSL Labs*[21] also offers browser-based tests for SSL clients that display a very concise report of its TLS capabilities, intended for the expert user. In BrowserAudit we instead strived to produce reports that can be interpreted by users at different levels of technical competence.

The *CanIUse* test suite[22] gathers browser compatibility data for a wide variety of browser features such as support for HTML5 and CSS3. Some of these tests are automatic and others require visual confirmation or interaction from the user. A few tests check for support of security features. For example, one (interactive) test detects support for the CSP. In contrast, BrowserAudit runs around 220 automatic tests to assess the security of the whole CSP implementation.

The *DOM access checker*[23] is a web page also included in the Chromium browser source code that uses JavaScript to test the enforcement of some domain-related security policies such as cross-domain DOM access, JavaScript cookies, XMLHttpRequest calls, and event and transition handling. For example, it runs hundreds of tests to ensure that read or write attempts to the visible properties of the `document` object are blocked cross-domain. In contrast, we are satisfied with testing cross-domain access for one representative property of the `document` object: if such access is blocked, we conclude that the policy is effective. We could programmatically extend our tests to try accessing all properties, but that goes beyond the scope of our prototype: DOM-based cross-domain access is only one of the hundreds of qualitatively different behaviours that we consider.

Finally, the project closest to ours is *Browserscope*,[24] a community-driven project for profiling web browsers, which detects the browser version and runs tests that cover a broad range of features such as network performance, CSS support and, most interestingly, security. Test results are aggregated and made publicly available, making it easy for web developers to keep track of functionality across all browsers that have been tested. *Browserscope* currently includes 17 browser security tests which are run automatically in the browser, in a similar fashion to ours, and that cover a limited number of standard security features that should be correctly implemented in each major browser. Our goals are broadly aligned with those of the Browserscope security test suite,[25] but cover a substantially larger set of features. Moreover, since Browserscope focusses on profiling, its output is simply a list of which tests passed or failed, with no indication of why. In contrast, we provide detailed descriptions of the main security features and show how each case is being tested, along with the expected and actual results.

6. CONCLUSIONS

We introduced BrowserAudit, a web application to test the implementation of browser security features. It complements the unit testing used by browser vendors to debug their implementations by checking that deployed browsers effectively deliver the security behaviours entailed by the specifications of browser security mechanisms.

All of our tests run automatically without interaction from the user, and provide detailed information for each test category, including the source code of each individual test. This makes BrowserAudit useful for a broad audience, from the casual user to the web developer and the security researcher. No other publicly-accessible web application tests such a breadth of browser security mechanisms as ours, either established or experimental.

BrowserAudit is designed to be modular and easily extensible with new tests. In Section 4.3 we highlighted parts of the browser security mechanisms currently not covered. We are planning to open-source BrowserAudit and hope to enlist help from the web security community to extend it with even more test cases.

7. REFERENCES

[1] D. Akhawe, P. Saxena, and D. Song. Privilege Separation in HTML5 Applications. In *Proceedings of*

[18] `https://panopticlick.eff.org/`

[19] `http://browserspy.dk/`

[20] `https://www.howsmyssl.com/`

[21] `https://www.ssllabs.com/`

[22] `http://tests.caniuse.com/`

[23] `http://lcamtuf.coredump.cx/dom_checker/`

[24] `http://www.browserscope.org/`

[25] `http://mayscript.com/blog/collinj/what-makes-good-browserscope-security-test`

USENIX Security 2012, pages 429–444, 2012.

[2] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), Apr. 2011.

[3] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 360–371, Washington, DC, USA, 2009. IEEE Computer Society.

[4] A. Barth, C. Jackson, and J. Mitchell. Securing Frame Communication in Browsers. In *Proceedings of USENIX Security 2008*, pages 17–30, 2008.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-site Request Forgery. In *Proceedings of CCS'08*, pages 75–88, 2008.

[6] A. Barth and M. West. Content Security Policy 1.1, June 2013. W3C Working Draft WD-CSP11-20130604.

[7] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-Based Defenses Against Untrusted Browser Origins. In *Proceedings of USENIX Security 2013*, pages 653–670, 2013.

[8] L. Bilge, T. Strufe, D. Balzarotti, and E. Kirda. All Your Contacts Are Belong to Us: Automated Identity Theft Attacks on Social Networks. In *Proceedings of WWW'09*, pages 551–560, 2009.

[9] E. Budianto, Y. Jia, X. Dong, P. Saxena, and Z. Liang. You Can't Be Me: Enabling Trusted Paths and User Sub-origins in Web Browsers. In *Proceedings of RAID 2014*, pages 150–171, 2014.

[10] P. Chen, N. Nikiforakis, C. Huygens, and L. Desmet. A dangerous mix: Large-scale analysis of mixed-content websites. In *Proceedings of ISC 2013*, 2013.

[11] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang. Lightweight Server Support for Browser-based CSRF Protection. In *Proceedings of WWW'13*, pages 273–284, 2013.

[12] P. De Ryck, L. Desmet, P. Philippaerts, and F. Piessens. A security analysis of next generation web standards. Technical report, ENISA, July 2011.

[13] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting sensitive web content from client-side vulnerabilities with CRYPTONS. In *Proceedings of CCS'13*, pages 1311–1324, 2013.

[14] P. Eckersley. How unique is your web browser? In *Proceedings of PETS'10*, pages 1–18, 2010.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[16] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In *Proceedings of CCS'12*, pages 760–771, 2012.

[17] I. Hickson and D. Hyatt. HTML5: A vocabulary and associated APIs for HTML and XHTML. W3C Candidate Recommendation CR-html5-20140429, Apr. 2014.

[18] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), Nov. 2012.

[19] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and Defenses. In *Proceedings of USENIX Security 2012*, pages 22–22, 2012.

[20] C. Jackson and A. Barth. Forcehttps: Protecting High-security Web Sites from Network Attacks. In *Proceedings of WWW'08*, pages 525–534, 2008.

[21] E. Kirda. Cross Site Scripting Attacks. In *Encyclopedia of Cryptography and Security*, pages 275–277. 2011.

[22] S. Maffeis, J. C. Mitchell, and A. Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *Proceedings of S&P 2010*, pages 125–140, 2010.

[23] L. Meyerovich, A. P. Felt, and M. Miller. Object Views: Fine-Grained Sharing in Browsers. In *Proceedings of WWW'10*, pages 721–730, 2010.

[24] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In *Proceedings of CCS'12*, pages 736–747, 2012.

[25] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *Proceedings of S&P 2013*, pages 541–555, 2013.

[26] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang. Towards Fine-Grained Access Control in JavaScript Contexts. In *Proceedings of ICDCS'11*, pages 720–729, 2011.

[27] D. Ross and T. Gondrom. HTTP Header Field X-Frame-Options. RFC 7034 (Informational), Oct. 2013.

[28] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting Framebusting: a Study of Clickjacking Vulnerabilities at Popular Sites. In *Proceedings of W2SP 2010*, 2010.

[29] T. Scholte, D. Balzarotti, and E. Kirda. Have Things Changed Now? An Empirical Study on Input Validation Vulnerabilities in Web Applications. *Computers & Security*, 31(3):344–356, May 2012.

[30] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of S&P 2010*, pages 463–478, 2010.

[31] B. Sterne and A. Barth. Content Security Policy 1.0. Nov. 2012. W3C Candidate Recommendation CR-CSP-20121115.

[32] M. Tran, X. Dong, Z. Liang, and X. Jiang. Tracking the Trackers: Fast and Scalable Dynamic Analysis of Web Content for Privacy Violations. In *Proceedings of ACNS 2012*, pages 418–435, 2012.

[33] T. Van Goethem, P. Chen, N. Nikiforakis, L. Desmet, and W. Joosen. Large-scale security analysis of the web: Challenges and findings. In *Proceedings of TRUST 2014*, pages 110–125, 2014.

[34] A. Van Kesteren. Cross-origin Resource Sharing. W3C Recommendation REC-cors-20140116, Jan. 2014.

[35] J. Wang, X. Li, X. Liu, X. Dong, J. Wang, Z. Liang, and Z. Feng. An Empirical Study of Dangerous Behaviors in Firefox Extensions. In *Proceedings of ISC 2012*, pages 188–203, 2012.

[36] M. Zalewski. Browser Security Handbook, 2010.

[37] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.