Department of Computing
Imperial College London

BrowserAudit

A web application that tests the security of browser implementations

Charlie Hothersall-Thomas

Supervisor: Dr. Sergio Maffeis

June 2014

Submitted in part fulfilment of the requirements for the degree of
Bachelor of Engineering in Computing

Abstract

The World Wide Web now reaches billions of users worldwide. It is used to run a wide range of applications for which security and privacy are essential. Web security is difficult. Even when a web developer follows all best security practices when developing and hosting his application, this is not always enough: he must still rely on his users to use a secure web browser when accessing his application. Whilst all web developers must assume that their users' browsers implement standard security features that have existed for many years, many of today's web applications also make use of modern security features that are also important.

The web browser world is incredibly diverse. Given any browser, it is difficult to determine precisely which browser security features it correctly implements. We explore a novel approach to browser security testing by automatically assessing the security of a browser with a web application. The resulting application, BrowserAudit, is able to test the security features of a browser by carefully combining the server- and client-side of the application. This means that we can assess the security of a browser by visiting a webpage. We face two key challenges in this project. Firstly, given a browser security feature, we must determine how best to test its implementation. We must then find a method to automatically test the feature in a way that works in all major browsers, with no input or interaction required from the user.

Our web application is able to automatically assess the security of most browsers with good accuracy. As a proof of its usefulness, BrowserAudit identifies two security bugs in the latest version of Mozilla Firefox. Looking towards the future, we propose many additional security features that BrowserAudit could test, as well as discussing the possibility of open sourcing the application to invite other developers to build upon our existing test coverage.

Acknowledgements

I would like to thank my supervisor, Dr. Sergio Maffeis, for proposing the project and providing me with constant support, motivation and inspiration throughout. His feedback on a draft version of this report was invaluable. I am also grateful that Sergio was happy to provide funding for the long-term hosting of the application, as well as the necessary domain names and SSL certificates. Thanks also to my family and friends (both those at Imperial College London and those from home) for their ongoing and unconditional support throughout my time here.

The Web as I envisaged it, we have not seen it yet. The future is still so much bigger than the past.

– Sir Tim Berners-Lee, 2009

Contents

1 Introduction

1.1 Motivation

The World Wide Web began as an exclusive system, used by just a handful of technically-minded individuals to share simple webpages about themselves and their work with other users around the world. Just a little over twenty years later, the Web now represents a medium reaching billions of casual users worldwide. The Web's rate of growth was phenomenal; it has evolved into a widely-used platform not just for simple static content, but for a wide range of complex interactive applications.

The Web originally consisted entirely of static content: simple pages of hypertext that could link between each other. We then started producing these pages on demand, customising the contents of the page for each individual user. With a 1995 release of the Netscape Navigator browser came JavaScript [28], a client-side scripting language that is used extensively today to add even more interactivity to webpages. We have *XMLHttpRequest*, an API allowing asynchronous HTTP communications between client-side JavaScript and the server without the need to load a new webpage. The introduction of this API into JavaScript resulted in a new wave of responsive web applications – called "Web 2.0" at the time – further contributing to the growth of the Web.

With the Web's growth came an increased need for security. The platform is currently being used to run more and more applications in which security and privacy are essential. Such applications include online banking, cloud storage of documents and photos, and private communications such as email clients and instant messaging. In the last two years, cryptocurrencies such as Bitcoin have gained popularity, with plenty of web applications arising to facilitate the storage and spending of these digital funds. When the Web was first proposed, no one could have predicted just how big it would become, and hence not as much consideration was paid towards security as would be if the entire system were to be redesigned from scratch today. Whilst lots can be done on the server-side of a website to ensure its security, web application developers must also rely on the web browsers used by their visitors to access their websites to correctly implement various standard security features. Without these security features, the Web as we use it today could not be considered secure.

Many web browser features are implemented first and standardised afterwards. Experimental implementations are rolled out first; keen web developers then begin using these features immediately, at which point other browser vendors may choose to copy the feature and write their own implementation. Eventually, if the feature becomes popular enough, it may be standardised formally by a body such as the World Wide Web Consortium (W3C). By this point, however, it is often already the case that there is a lack of standardisation across browsers. The fact that many browser features begin as experimental implementations can also lead to undesirable designs that can't be changed. This pattern perhaps began as early as 1994, when Netscape's first implementation of web cookies led to a scenario in which there was no accurate or official account of cookie behaviour in any modern browsers [41]. The four-page proposal of a single Netscape engineer [30] quickly became the de facto standard for cookies, and laid the heavy foundations for how we use cookies today. Many developers have written about problems in the design of the cookie system we use today, and a "Cookies 2.0" specification was even proposed [26], but the current system is far too widely adopted for it to be changed now. More recently, a browser security feature known as the Content Security Policy was born in experimental implementations. As such, in order to achieve maximum compatibility across all browsers, a developer must now use *three* different HTTP headers to achieve the same thing: `Content-Security-Policy`, `X-Content-Security-Policy`, and `X-WebKit-CSP`. All of this is a side-effect of the Web's rate of growth, and can make it very difficult to keep track of and standardise browser security.

The web browser world is highly diverse: there exist many different browsers, of which there are multiple versions covering a myriad of operating systems and hardware platforms. The big five are Mozilla Firefox (which evolved from the Netscape Navigator browser mentioned above), Google Chrome, Internet Explorer, Safari, and Opera. There are many derivatives of these, however, based on the underlying engines such as WebKit and Chromium. There are also many applications in the wild that embed customised browsers, for example Valve's popular Steam gaming client uses the Chromium Embedded Framework to render web content [11]. Some ATMs have been seen in the wild using Internet Explorer 6! An incredibly old version of the browser which wouldn't be considered secure by many today, yet is seemingly responsible for handling cash transactions. Each browser vendor is responsible for ensuring the security of their product across all such platforms but, at present, there is no good way to confirm whether or not a given browser has a complete set of standard security features correctly implemented.

There are still browsers in use today that do not implement the security features that we would expect a modern-day browser to implement. According to current Google Play Store statistics, 20.0% of devices running the Android platform use Android 2.3 Gingerbread [12]. This is unsettling, since Gingerbread's default browser fails to implement many standard security features such as the *HttpOnly* cookie flag and Strict Transport Security [19]. The lack of an *HttpOnly* cookie implementation is especially hard to believe, since it is a security feature that has been implemented in all major browsers for many years [22].

1.1.1 Objectives

The goal of this project is to create a web application, accessible primarily as a webpage, that tests the security features of the browser used to access the page. The security tests will run automatically, requiring minimal interaction from the user. The application will produce a comprehensive report for the user of those features that are implemented correctly, and those that are not and therefore contain vulnerabilities. There are a number of beneficiaries who could make use of this application, namely browser developers, security researchers, penetration testers, and even security-conscious users. Provided that the application front-end isn't overwhelmingly technical, our application could also be used by more typical web users to educate them on browser security and the importance of keeping their browser(s) up to date.

We plan to achieve this by having the server-side of our application make use of as many browser security features as possible. The client-side of our application will then be used to ensure that these features are correctly implemented and report any issues to the user. This is an interesting problem because, for each security feature, we must come up with a means of testing it. In addition to this, we must implement the tests so that they run automatically with no input required from the user. All of this must be achieved whilst ensuring that our implementation works in as many different web browsers as possible.

1.2 Contributions

We have produced a web application, BrowserAudit, that automatically tests the implementations of various security features in the browser used to access it. BrowserAudit automatically executes over 300 tests. We chose to implement a selection of tests that covers both the most important browser security concepts (features that should be implemented in any browser) and modern security features that are not yet implemented by all of today's major browsers. We have designed the user interface of our application so that it can educate web developers and security-conscious web users about the security features we test. We provide textual descriptions of each security feature, and hope that by showing each individual test result, a user will be able to gain a better understanding of each security feature.

BrowserAudit has been developed in a way that allows new tests to be added in a simple and modular manner. We consider the possibility of releasing the source code to the public shortly, inviting like-minded developers to build upon our current codebase to test even more browser security features.

We have been able to evaluate the usefulness of our project already, since it identifies two security bugs in the latest version of Mozilla Firefox, a popular browser with 20.4%

usage share for desktop browsers as of January 2014 [35]! We reported the bugs to the Firefox development team, who promptly accepted them as valid bugs.

Our application is online today and can be accessed at `https://browseraudit.com/`.

1.3 Report Structure

The remainder of this report is separated into six chapters:

- Background (page 5) — we cover the technical background necessary to explain how our browser tests work, by first discussing the basics of HTTP and then describing various browser security features in detail. We also discuss related work discovered during our background research for the project;

- Design (page 37) — we discuss the implementation languages chosen and the key libraries used. We showcase our front-end website design with screenshots. We also describe our server-side system architecture;

- Implementation (page 50) — general implementation topics. We discuss how our custom web server is implemented and important common code that is relevant to each of our browser tests;

- Browser Tests (page 61) — we explain how we are able to automatically test the browser security features described in the Background chapter, and problems encountered along the way. This involves describing the implementations of the tests, supplemented with code listings and diagrams where appropriate;

- Evaluation (page 106) — we evaluate the success of our project. This includes discussing limitations of BrowserAudit in its current state, as well as assessing the project through primarily qualitative metrics;

- Conclusions (page 117) — closing thoughts. The majority of this chapter is the Future Work section, in which we propose various improvements that could be made to the project in the future.

2 Background

In this chapter we first describe the basics of the Web, focusing primarily on the hypertext transfer protocol. We also use this background chapter to describe in detail the security features that BrowserAudit will test. This is necessary in order to understand how we test the features later on in Chapter 5. In Section 2.3 we discuss related work discovered during our background research for the project.

2.1 Web Basics

In this section we describe the basics of how the Web works. This is important in order for us to be able to describe how various browser security features work in Section 2.2, and how we test them in Chapter 5.

Although many people refer to the Web and the Internet synonymously, this is not correct. The Web is just one of many applications running on the Internet. Other common examples of Internet applications include email and DNS (domain name system).

2.1.1 Hypertext Transfer Protocol

The Web uses the hypertext transfer protocol (HTTP) for requests and responses. HTTP is an application protocol at level 7 of the OSI networking model. It presumes an underlying and reliable transport layer protocol, and is most commonly used with TCP. The Web typically uses HTTP on port 80 for unencrypted requests and port 443 for secure (HTTPS) requests which are discussed in more detail in Section 2.1.2.

HTTP is a request/response protocol with a client and a server. In the case of the Web, a web browser is the client and makes requests to a web server. The server replies to these requests with responses. For example, a browser might send a request for a specific webpage `http://example.com/index.html`. The server's response will include the contents of the document as well as other information about the request. An example of how this request might look is shown in Listing 1.

```
1  GET /index.html HTTP/1.1
2  Host: example.com
3  Accept: */*
```

```
1  HTTP/1.1 200 OK
2  Server: nginx/1.4.6 (Ubuntu)
3  Date: Sun, 01 Jun 2014 13:31:56 GMT
4  Content-Type: text/html
5  Content-Length: 164
6  Last-Modified: Sun, 01 Jun 2014 13:31:34 GMT
7  Connection: keep-alive
8  Vary: Accept-Encoding
9  ETag: "538b2b36-a4"
10 Accept-Ranges: bytes
11
12 <!DOCTYPE html>
13 <html lang="en">
14   <head>
15     <meta charset="utf-8" />
16     <title>Test Page</title>
17   </head>
18   <body>
19     <h1>Hello, world!</h1>
20   </body>
21 </html>
```

Listing 1: An example HTTP request and response for an HTML webpage

In line 1 of the request we can see that the client is making a `GET` request for `/index.html`. It also states that it is using version 1.1 of the HTTP protocol. Here, `GET` is known as a *request method* (sometimes also known as HTTP verbs). `GET` is the most common of these methods, and requests a representation of the specified resource. The other two most common methods are `HEAD` and `POST`. `HEAD` requests a response identical to the one that would be returned after a `GET` request, but without the response body. This means that the server will return only the response headers, i.e. lines 1–10 in Listing 1. The `POST` method is used to request that the server stores some data enclosed in the request's body. For example, when one submits a registration form on a website, the form data will likely be transmitted in a `POST` request. The three verbs discussed so far are the only methods defined in the HTTP 1.0 specification [5]. HTTP 1.1 saw the introduction of many new methods, including an `OPTIONS` method to request the methods supported by the server for a given URL [15].

Request Headers

The remainder of the HTTP request (lines 2–3 in Listing 1) consists of **request headers**. We can see the `Host` header which specifies the domain name (and possibly port) of the server. This header exists so that one server can serve multiple domains, and is mandatory since HTTP 1.1. This is the only mandatory request header. There are many other request headers; some of the more common are:

- `Accept` – acceptable MIME content types (e.g. `text/plain`);
- `Accept-Encoding` – acceptable encodings, used for data compression (e.g. `gzip, deflate`);
- `Accept-Language` – acceptable content languages (e.g. `en-gb`);
- `Cookie` – a cookie stored by the browser that is relevant to the request (we discuss cookies on page 8);
- `Origin` – the origin of a request, used in cross-origin resource sharing (see Section 2.2.3);
- `Referer` – the address of the webpage that linked to the resource being requested. The name of this header was originally a misspelling of ‘referrer’;
- `User-Agent` – the user agent string, which can be used by the server to identify the browser being used to make the request.

Response Headers

In lines 2–10 of the HTTP response in Listing 1, we can see examples of HTTP **response headers**. These tell the client more information about the response. Some of the more common request headers are:

- `Cache-Control` – tells all caching mechanisms between the server and client (including the client) whether they may cache the object and for how long, measured in seconds (e.g. `max-age=300`);
- `Content-Language` – the (human) language of the content;
- `Content-Type` – the MIME content type of the response;

- `Set-Cookie` – sets an HTTP cookie for the browser to store.

Uniform Resource Locators

A uniform resource locator (URL) identifies a specific resource on a remote server. Commonly referred to as a web address, it is usually displayed prominently in a web browser's user interface. The URL syntax is detailed in RFC 3986 [6]; there are many optional elements, but a good working example is as follows:

```
scheme://host:port/path?query_string#fragment
```

The schemes, otherwise referred to as protocols, used most commonly by web applications are `http:` and `https:`. Other examples of schemes are `ftp:` and `file:`, and pseudo-URLs that begin with `data:` and `javascript:`.

The host is most commonly a domain name (e.g. `example.com`) but can also be a literal IPv4 or IPv6 address. If not otherwise specified, the port defaults to the port associated with the scheme (80 for `http:` and 443 for `https:`).

The path is used to specify the resource being accessed. The query string parameters contain optional data to be passed to the software running on the server. The fragment identifier, also optional, specifies an exact location within the document. In HTML documents, these fragment IDs are often combined with anchor tags to allow hyperlinks to specific sections within a document.

The most important elements of a URL as far as we are concerned are the scheme, host and port. We will see later on that, together as a tuple, they form a concept known as an *origin* used in many browser security concepts.

Cookies

Cookies are pieces of data stored on a user's computer by webpages they visit. When a browser requests a webpage from a server, it sends any cookies it has for that page with the request. Cookies are often used to store session information (for example, so that a user stays logged in-between browsing sessions), configuration settings (such as a preferred language), and sometimes even data for tracking a user's browsing habits.

A browser sends a cookie with its request using the `Cookie` header, whilst a server informs a client to save or overwrite a cookie with the `Set-Cookie` header. Below is an example `Set-Cookie` header that might be sent by a server:

```
Set-Cookie: foo=bar; Path=/; Domain=example.com;
  Expires=Tue, 01 Jul 2014 16:39:25 UTC
```

This sets the value of the `foo` cookie to be `bar`. In theory, multiple cookies can be set in a single `Set-Cookie` header. In practice, however, most browsers only support a single cookie per header. The browser will then send the cookie back with subsequent requests for the same application (as defined by the *Domain* and *Path* attributes – see Section 2.2.1) with a `Cookie` header like below:

```
Cookie: foo=bar
```

Unlike the `Set-Cookie` header, most browsers support the sending of multiple cookies in a single `Cookie` header, separated by semicolons.

The *Expires* attribute tells the browser the exact time when it should delete the cookie. Alternatively, although not as commonly used, the *Max-Age* attribute may be used to specify when the cookie should expire as a number of seconds in the future from when the browser receives the cookie. If no expiry time is set using either method, the cookie is termed a session cookie and is to be deleted when the browser is closed.

A cookie can be overwritten by sending a new `Set-Cookie` header, as long as the *Domain* and *Path* attributes are identical. To delete a cookie, a new `Set-Cookie` response header for a cookie with that name may be sent specifying an expiry date in the past. Again, it is important to ensure that the *Domain* and *Path* attributes are the same as the original cookie. To understand why, consider the case where the server sets two cookies with the same name:

```
Set-Cookie: foo=bar; Path=/; Domain=example.com
Set-Cookie: foo=baz; Path=/account; Domain=example.com
```

When sending a request to `http://example.com/account`, the browser now has two different cookies `foo=bar` and `foo=baz` that are relevant to the request according to the *Domain* and *Path* attributes. Which should it send with a `Cookie` header in the request? The answer is that a browser will send both cookies to the server, although there is no convention for the order in which a browser should send the conflicting cookies. The consequence of this is that the server will have no means of resolving the conflict – all it knows is the two values of the cookie, since the browser does not send any information about the age of the cookies in the `Cookie` header. Additional metadata to solve this problem is proposed in RFC 2965 [26] (the “Cookies 2.0” specification referred to in our introduction), but the standard did not become widely adopted.

There are two further cookie attributes, *HttpOnly* and *Secure*. These are discussed as browser security features in Section 2.2.4.

2.1.2 HTTP Secure

In HTTP as we have seen it so far, everything is transmitted over the wire in plaintext—both requests and responses, including any cookies and the documents themselves. Whilst this may not have been a concern in the early days of the Web, it holds many security and privacy consequences today. Wiretapping is a common threat, which could be used not only to spy on a user but also to steal his session cookies or other sensitive data. Man-in-the-middle attacks, in which an attacker pretends to be the intended web server and modifies a victim's requests/responses on the fly, are also incredibly easy to pull off as long as everything is transmitted in plaintext.

The solution to this is HTTP Secure (HTTPS). Proposed in RFC 2818, it suggests the idea of running HTTP over transport layer security [31]. Transport layer security, often referred to as TLS or SSL, is a cryptographic protocol for secure communications over the Internet. It had already been developed many years before the HTTPS RFC. With TLS, public key cryptography is used to both *encrypt* and *authenticate* the communication between two endpoints. HTTP can be run on top of this without the need for any changes to be made. With HTTPS, the problems of both wiretapping and man-in-the-middle attacks are solved. Any attacker viewing a victim's data over the wire cannot read the HTTP requests and responses due to the encryption[1]. A man-in-the-middle attack is not possible either since a web browser is able to authenticate the identity of the server with which it is communicating.

The authentication works using X.509 certificates which associate a public key with an identity. In the case of HTTPS, the certificates associate a public key with one or more domain names. These certificates are issued and cryptographically signed by *certificate authorities* (CAs). In order for this to work, browsers are shipped with a large set of public keys for the CAs that the browser vendors trust. The browser vendors trust these CAs to verify that a public key belongs to a particular website. When establishing a new HTTPS connection, the browser receives the certificate from the server. This certificate will have been signed by a CA – a signature that cannot be produced without their private key. The browser verifies the signature by following the trust chain to one of the CA public keys hardcoded in the browser. As well as checking that the certificate is trusted, the browser also checks that it is valid for the website being visited. It does this by checking that the site's domain matches one of the domains in the certificate. Finally, the browser verifies that the certificate has not been included on any public revocation lists (which might be the case if, for example, a private key had been lost or compromised). If all of these tests pass, the browser can then begin sending encrypted communications to the server knowing that only the intended recipient will be able to decrypt them.

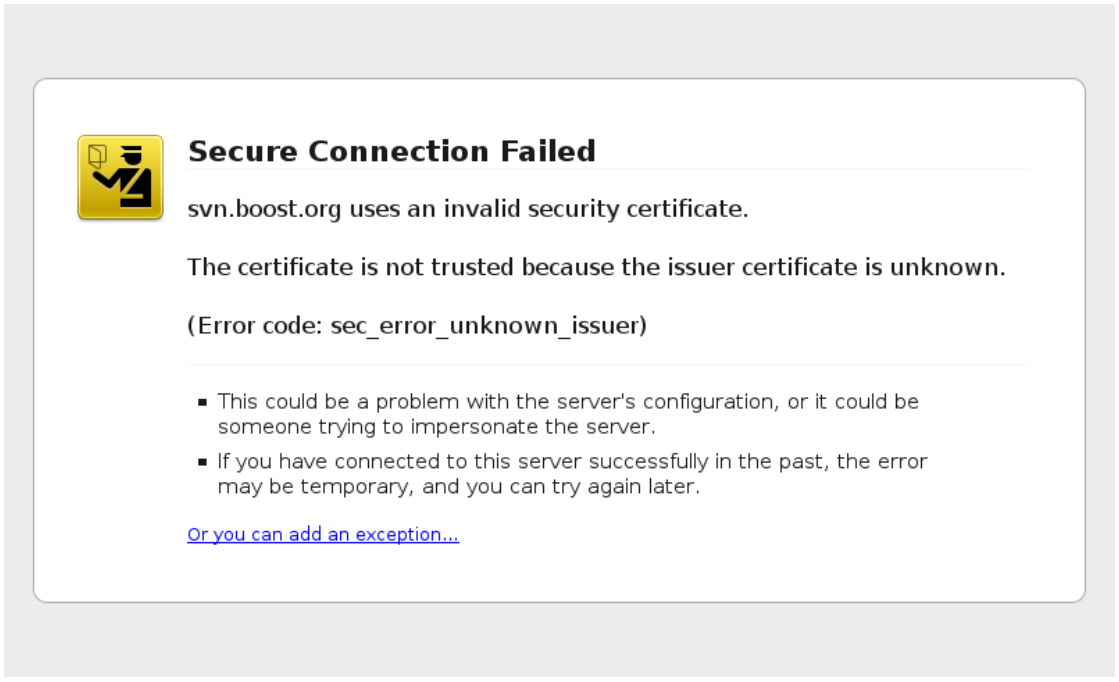
[1]Whilst the HTTP communication is encrypted, a wiretapping attacker may still be able to learn the websites being viewed by his victim. This is because the DNS protocol used to resolve a domain name to an IP address is not usually encrypted

Session Keys and Perfect Forward Secrecy

TLS uses the long-term public and private keys to exchange a short-term session key that is used to encrypt the data for the rest of the session. This session key is used in symmetric cryptography, since this is less resource-intensive than asymmetric (public key) cryptography. An important property here is **perfect forward secrecy** (PFS). Perfect forward secrecy means that if a long-term private key were to become compromised in the future, the short-term session key used in a previous HTTPS session cannot be derived because of this. Usage of PFS has increased recently but is far from widespread – support for it is still lacking in two of the five major browsers [13].

Self-Signed Certificates

It is possible for a website to sign its own SSL certificate, claiming that its public key belongs to its domain. Since the browser will not trust the website as a certificate authority, this will lead to a warning in most web browsers. This is because the identity of the server cannot be identified by the browser. An example warning in Mozilla Firefox is shown in Figure 2.1.



Secure Connection Failed

svn.boost.org uses an invalid security certificate.

The certificate is not trusted because the issuer certificate is unknown.

(Error code: sec_error_unknown_issuer)

- This could be a problem with the server's configuration, or it could be someone trying to impersonate the server.
- If you have connected to this server successfully in the past, the error may be temporary, and you can try again later.

Or you can add an exception…

Figure 2.1: A warning in Firefox caused by a self-signed SSL certificate

The warnings can usually be ignored by the user, except in the case of HTTP Strict Transport Security, a relatively young security feature which we visit in Section 2.2.5.

2.2 Browser Security Features

In this section we cover the technical background of many browser security features that exist to secure the Web. Some of these are absolute must-have features that should be implemented in any browser, whereas others are modern features that have appeared in some but not all browsers in recent years. This technical background is necessary to understand how BrowserAudit automatically tests these features in Chapter 5.

2.2.1 Same-Origin Policy

A big aspect of browser security is content isolation: the browser promises to isolate documents based on where they came from. The concept of the same-origin policy is arguably the most important idea in web browser security. The idea is a simple one: two webpages served by the same *origin* should be able to interact mostly unconstrained, whereas two webpages served by different origins should not be able to interfere with one another at all. We define an origin as a `(scheme, host, port)` tuple [3], with the major exception that Internet Explorer ignores the port when comparing origins. Table 2.1 gives example results of origin comparisons to the URL `http://login.example.com/a/page.html`.

URL	Outcome	Reason
`http://login.example.com/b/other.html`	Success	
`http://login.example.com/a/dir/another.html`	Success	
`https://login.example.com/secure.html`	Failure	Scheme mismatch
`http://login.example.com:81/a/contact.html`	Failure	Port mismatch
`http://payments.example.com/a/contact.html`	Failure	Host mismatch

Table 2.1: Example origin comparisons when comparing to `http://login.example.com/a/page.html`

Without the same-origin policy, a user's session with an honest website could be interfered with as a result of the user visiting a dishonest website. This is exactly what the same-origin policy was designed to prevent, by isolating the content of each individual webpage open in a browser. Note that there is no single same-origin policy; the term instead refers to a set of related browser security policies, some of which we discuss below. The same-origin policy should definitely be implemented in all browsers, since it is such an important web security feature.

DOM Access

The Document Object Model (DOM) is an API for HTML and XML documents. It is a structured representation of a document that allows a developer to modify its content or appearance. The DOM is most commonly used in JavaScript, accessed by the `document` object. There are many DOM methods and properties that can be accessed. As a trivial example, the JavaScript in Listing 2 shows how the DOM can be used to modify the `src` attribute of an image `myImage` on a webpage.

```
var image = document.getElementById("myImage");
image.src = "newPicture.png";
```

Listing 2: Using the DOM to modify an image's `src` attribute

It should be clear that no website loaded in a browser should be able to access or modify the DOM of another. The same-origin policy for DOM access ensures that this is the case. The origins of the original document and the accessed object are compared. If they are equal then the browser allows access with no further checks. One might expect that if they don't match then access is immediately denied, however there is one further step in the decision-making process thanks to the `document.domain` property.

`document.domain` allows two cooperating webpages to lift the same-origin policy restrictions, provided both webpages agree to do so. For example, if a page at `http://example.com` contains a frame with content from `http://login.example.com`, the same-origin policy would ordinarily block either page from accessing the DOM of the other due to the mismatching hostnames. However if they both set their `document.domain` property to `example.com`, the same-origin policy restrictions will be lifted by the browser and access will be allowed.

The use of the `document.domain` property as described above is not without its own restrictions. The most important rule is that, when a page sets the value, the new value must be a right-hand, fully-qualified fragment of its current hostname [40]. This means that a page at `http://login.secure.example.com` may set its `document.domain` property to `secure.example.com` or `example.com` but not `staff.secure.example.com` or `ample.com`. The other rule is that both webpages must set the property to the same value in order for DOM access between them to be granted – it is not sufficient for a page at `http://login.example.com` to set its `document.domain` property to `example.com` in order to access the DOM of a page at `http://example.com`. For this to work, the page at `http://example.com` must also explicitly set its domain property to `example.com`. The technical explanation for this is that the port number of each page is kept separately by the browser. When the `document.domain` property is written to, this causes the port to be overwritten with *null* [33]. This means that setting the property on one page but not the other will result in one document having port *null* and the other its original value (e.g. 80). The origins will then still mismatch, despite both documents appearing to have the same host. When both pages set their `document.domain` property, each page will have port *null* and so access will be allowed provided the domains and schemes are the same.

If the origins of two webpages do not match, and they don't define the same (legal) `document.domain` property, then DOM access between them should be denied by the browser.

XMLHttpRequest

XMLHttpRequest, as mentioned in our Introduction chapter, is an API whose introduction into JavaScript was responsible for the appearance of a new wave of responsive web applications referred to by the industry as "Web 2.0". The API allows a developer to easily retrieve data from a URL without the need for a full page reload. This means that a small section of a webpage can be updated on the fly with no interruption to the user. Better still, this can be done asynchronously, so that the *XMLHttpRequest* calls are not blocking. This is known as AJAX, asynchronous JavaScript and XML, and is employed in many of today's most popular web applications. In fact, we make heavy use of AJAX in BrowserAudit! It is necessary for our client-side testing scripts to retrieve data from the server-side.

The same-origin policy applies to *XMLHttpRequest* in a very similar manner to the DOM. That is to say, a client-side script may only make HTTP requests by *XMLHttpRequest* for documents with the same origin that it came from. There are two key differences when comparing the same-origin policy for *XMLHttpRequest* to the one for the DOM:

- the `document.domain` property has no effect on origin checks for *XMLHttpRequest* targets, meaning it is not possible for cooperating websites to agree for there to be cross-domain requests between them in this way;
- Internet Explorer takes the port number into account when comparing origins for *XMLHttpRequest* targets.

In Section 2.2.3 we look at cross-origin resource sharing, a security mechanism that allows cross-domain requests using *XMLHttpRequest*.

Cookies

There are many aspects to cookie security, however for now we will only discuss how the same-origin policy applies to cookies. It does so through one fairly simple concept: *scope*. The scope of a cookie defines which cookies a browser will send with each request that it makes, to ensure that it only sends cookies that should be relevant to the receiving server. We wouldn't want our browser to send our online banking authentication cookie to a website hosting free Flash games, for example!

The scope of a cookie comprises of two attributes: *Domain* and *Path*. These are set by the server in the `Set-Cookie` header. If not set, they default to the domain and path of the requested object that is being returned with the `Set-Cookie` header [2].

The *Domain* parameter may be used to *broaden* the scope of a cookie. It can be set to any fully-qualified right-hand segment of the current hostname, up to one level below the TLD[2]. This means that a page at `payments.secure.example.com` may tell the browser send a cookie to `*.secure.example.com` or `*.example.com`, but not to `www.payments.secure.example.com` (since this is more specific than the cookie origin) or `*.com` (since this is too broad). To set a cookie to be sent to `*.example.com`, the *Domain* parameter can be set to `.example.com` or `example.com` (in practice, the former will now be rewritten to be the latter by the browser). This has the interesting consequence that it is not possible to scope a domain to a specific domain only. The only way to achieve this is to not set the *Domain* parameter at all, but even this does not function as expected in Internet Explorer.

The *Path* parameter is used to *restrict* the scope of a cookie. It specifies a path prefix, telling the browser to send the cookie only with requests matching that path. The paths are matched from the left, so a cookie with a path of `/user` will be sent with requests to both `/user/status` and `/user/account`.

Note that the scheme and port are not relevant in cookie scope like they are in the same-origin policies for the DOM and *XMLHttpRequest*. Only the hostname and path affect a cookie's scope.

2.2.2 Content Security Policy

A cross-site scripting (XSS) attack is one in which an attacker injects malicious client-side script into a webpage. This can have devastating effects, for example an attacker may be able to access cookies containing session tokens or other sensitive information

[2]A top-level domain (TLD) is a domain at the highest level in the hierarchical DNS system, for example `.com` and `.net` are TLDs

to do with the webpage being viewed. He may also modify the HTML contents of the page, perhaps tricking the user into clicking a malicious link or otherwise acting on false information planted by the attacker. The **Content Security Policy** is a relatively new mechanism that can be used to mitigate XSS attacks.

The Content Security Policy (CSP) was introduced because it is otherwise impossible for a browser to tell the difference between benign client-side script that's part of a web application, and client-side script that has somehow been maliciously injected by an attacker. For example, many popular websites choose to load common JavaScript libraries from content delivery networks (CDNs) such as `ajax.googleapis.com`. This is fine, since the developers trust the Google CDN, but as far as a browser is concerned `ajax.googleapis.com` is just as trustworthy a script source as `evil.attacker.com`! XSS attack mitigation is difficult when a browser is willing to download and run scripts from any source.

The CSP aims to solve this problem with the concept of *source whitelists*. Through the introduction of a new HTTP response header, a web application developer can explicitly specify the sources from which scripts and other resources for his webpages may be loaded. This header is `Content-Security-Policy` and is employed by many of today's most popular websites. When the header is set, a compliant browser will only load resources from the sources specified in the header. If the page tries to load a resource from some other source, it will not be loaded and a security exception will be thrown. This provides defence in depth, heavily reducing the damage an attacker could do in an XSS attack.

The policy applies not just to scripts but to many different kinds of resources such as images, stylesheets, media (`<audio>` and `<video>` tags), frames, fonts and objects (`<object>`, `<embed>` and `<applet>` tags). The allowed sources for each of these resource types may be different, allowing a web developer to be as specific as possible about where different resources may come from. The policy also allows a default source to be set, which is used as fallback whenever the source for a particular resource type has not been explicitly specified. A common usage pattern is therefore to set the default source to `'none'` (stating that no resources may be loaded from anywhere) and then set individual allowed sources for each resource type used on the page. This exercises the principle of least privilege nicely, ensuring that only the required resources can be loaded.

The `Content-Security-Policy` header value is made up of one or more directives separated by semicolons. Some of the more common directives are `default-src` (which specifies the default policy for all resource types), `script-src`, `style-src` and `img-src`. Most directives contain a list of allowed sources for a resource type known as a *source list*.

Source Lists

Each source list can contain multiple source values separated by a space, except in the two special cases `*` and `'none'` which should be the only value. Some example source values are:

- `*` – wildcard, allows any source;
- `http://*.example.com` – allows loading from any subdomain of `example.com` and `example.com` itself using the `http:` scheme;
- `https://payments.example.com` – allows loading from `payments.example.com` using the `https:` scheme.

If no port number is specified, it defaults to the port associated with the specified scheme. When no scheme is specified, it is assumed to be the same scheme as the one used to access the document that is being protected. The CSP permits that a browser may follow redirects before checking the origin of a resource.

There are also four keywords that can be used in a source list:

- `'self'` – allows loading from the same origin (scheme, host, port) as the page being served;
- `'none'` – prevents loading from any source;
- `'unsafe-inline'` – allows the use of unsafe inline resources, such as `<script>` and `<style>` tags and `style`/`onload` HTML attributes;
- `'unsafe-eval'` – allows the use of `eval()` and other methods for executing strings of code.

The `'unsafe-inline'` and `'unsafe-eval'` keywords can open up a webpage to XSS vulnerabilities, and so they must be explicitly allowed by a developer. Wherever possible, these options should be avoided in favour of defining styles, scripts and event handlers in `.css` and `.js` files which should be included in the document and allowed by the Content Security Policy in the normal way.

Content-Security-Policy-Report-Only and report-uri

The `report-uri` directive specifies a URL to which a browser should send a report about any policy violation. This report is a JSON object containing useful information such as the document URL, the blocked URL and the CSP directive that was violated. The report is sent to the server with a `POST` request. This is useful for webmasters to learn about attempted attacks, or to learn about an error in their `Content-Security-Policy` header that is wrongly blocking a resource from being loaded. Note, however, that many anti-tracking browser extensions will block the report request from being sent.

There is also a completely different HTTP response header, `Content-Security-Policy-Report-Only`, that allows servers to experiment with the CSP by monitoring (rather than enforcing) a policy. This can be useful to test and fine-tune a policy before rolling it out on a large website and potentially blocking innocent resources from being loaded, causing interruptions to users.

Example: GitHub

```
Content-Security-Policy:  default-src
                            *;
                          script-src
                            'self'
                            https://github.global.ssl.fastly.net
                            https://ssl.google-analytics.com
                            https://collector-cdn.github.com
                            https://embed.github.com
                            https://raw.github.com;
                          style-src
                            'self'
                            'unsafe-inline'
                            https://github.global.ssl.fastly.net;
                          object-src
                            https://github.global.ssl.fastly.net
```

Listing 3: GitHub's usage of the Content Security Policy

Listing 3 shows GitHub's `Content-Security-Policy` response header, which has been separated onto multiple lines for readability purposes only. Note that the CSP is being used not just to whitelist JavaScript sources, but also stylesheets (`style-src`) and plugins (`object-src`). Whenever a page served with this header tries to load a script, stylesheet or plugin from a source not in the whitelist, a compliant browser will throw an error. Other resource types can be loaded from anywhere due to the wildcard value used in the `default-src` directive. We can also see how keywords in source lists can be combined with other keywords and allowed sources.

Browser Support

As with many web features, the Content Security Policy was introduced in multiple browsers before it became standardised. Earlier experimental implementations in Mozilla Firefox and Google Chrome used header names `X-Content-Security-Policy` and `X-WebKit-CSP` respectively instead of the now-standardised `Content-Security-Policy` header which the latest versions of each browser use today. Internet Explorer versions 10 and 11 (the only versions of IE to support the CSP) still use the `X-Content-Security-Policy` header rather than the new standardised header [9]. This has the consequence that, despite CSP now being standardised, a website must serve its pages with all three CSP headers in order to achieve maximum compatibility with old and new browsers.

Content Security Policy 1.1

The standardised Content Security Policy described in this section refers to the Content Security Policy 1.0 Candidate Recommendation [36]. There also exists a working draft for a Content Security Policy 1.1 [4] which promises some interesting new features such as whitelisting specific inline script blocks via either nonces or hashes, and the ability to inject the policy through `<meta />` HTML tags as opposed to specifying it in an HTTP response header.

2.2.3 Cross-Origin Resource Sharing

When discussing the same-origin policy for *XMLHttpRequest*, we saw that there is no way for two cooperating websites to allow cross-domain requests between them. This is because the `document.domain` property isn't used in same-origin policy checks for *XMLHttpRequest* and therefore can't be used to lift the browser's security restrictions like we saw for DOM access. There is a modern solution, however, known as **cross-origin resource sharing** (CORS).

CORS is an extension to the *XMLHttpRequest* API that allows a website to carry out cross-origin communications. This means that client-side JavaScript can send an *XMLHttpRequest* to a URL on a domain (or scheme or port) other than the one from which it originated – something that is not otherwise possible due to the same-origin policy, which has frustrated web developers for quite some time. As of January 2014, CORS has been formally specified by the W3C [39], however the implementations in the browsers that implement CORS today differ from the specification. CORS is supported in Chrome 3+, Firefox 3.5+, Opera 12+, Safari 4+ and Internet Explorer 10+ [10]. The differences between the implementations and the specification are highlighted below.

Cross-origin resource sharing allows a request to be made to a server at a different origin only if the server receiving the request explicitly allows it. That is, the server states whether or not the origin of the requesting document is allowed to make a cross-origin request to that URL. To achieve this, CORS defines a mechanism that allows the browser and server to know just enough about each other so that they can determine whether or not to allow the cross-origin request. This is primarily achieved by two key headers: an `Origin` header sent by the browser with the request, and an `Access-Control-Allow-Origin` header sent in the server's response. There are other CORS-related headers that we also discuss below.

When a webpage attempts to load a cross-origin document through the *XMLHttpRequest* API, a browser implementing CORS first decides whether the request is *simple* or *non-simple*. A simple request is defined as one that meets the following criteria:

- HTTP method matches (case-sensitive) one of:
  - `GET`
  - `HEAD`
  - `POST`
- HTTP headers match (case-insensitive):
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Last-Event-ID`
  - `Content-Type`, where the value is one of:
    - `application/x-www-form-urlencoded`
    - `multipart/form-data`
    - `text/plain`

A request is said to be non-simple if it does not meet the above criteria. This distinction is important because it defines how the request should be handled by the browser. Note, however, that the browsers implementing CORS today do not precisely follow the specification above. They instead ignore the above recommended whitelist of headers and treat any requests with user-defined header values as non-simple requests. In addition to this, the implementation in WebKit-based browsers (e.g. Safari) treats any requests containing a payload as non-simple [42].

Simple Requests

A simple request is sent to the server immediately. The request is sent with an `Origin` header containing the origin of the calling script. The retrieved data is only revealed to the caller if the server's response contains an acceptable `Access-Control-Allow-Origin` header. This header specifies an origin that is allowed to access the content. If this header is not set, or its value does not match the origin of the calling script, then the browser must not pass the retrieved data onto script that made the request. The header may only contain a single origin value; this may seem problematic if a URL is to be accessed cross-origin from multiple different origins, which is why the `Origin` header is sent by the browser. The server can look at the `Origin` header and dynamically decide which `Access-Control-Allow-Origin` header to send to allow the request if the origin is to be permitted. A wildcard value (`*`) is also valid, although rarely used in practice due to the security implications of allowing a document to be accessed with *XMLHttpRequest* by any origin.

Non-Simple Requests

Unlike simple requests, a non-simple request is not immediately sent to the server. A "preflight" request is instead sent to the destination server in order to confirm that it is CORS-compliant and happy to receive non-standard traffic from that particular caller. This preflight request is sent with the `OPTIONS` HTTP method with headers containing an outline of the parameters of the underlying *XMLHttpRequest* call. The most important information is sent in three headers:

- `Origin` – the origin of the calling script, just like is sent with a simple request (e.g. `https://browseraudit.com`);
- `Access-Control-Request-Method` – the HTTP method of the non-simple request the caller is trying to make (e.g. `PUT`);

- `Access-Control-Request-Headers` – a list of any non-simple headers[3] that are included in the caller's request, separated by commas (e.g. `X-My-Header,X-Another-Header`).

The two-step handshake is considered successful by the browser only if these three parameters are properly acknowledged in the server's response to the preflight request. The server acknowledges the parameters with the corresponding headers `Access-Control-Allow-Origin`, `Access-Control-Allow-Method` and, when relevant, `Access-Control-Allow-Headers`. The first two of these headers must be present – a response with no CORS headers is an error and so the non-simple request should not be made by the browser. Likewise, the handshake has failed if the headers are present but the values do not correspond to those sent by the browser in the preflight request. As with a simple request, the `Access-Control-Allow-Origin` header may only contain a single origin that is allowed to make the request – this must either be the wildcard value or match the origin of the calling script sent by the browser in the `Origin` header. The `Access-Control-Allow-Method` header may contain a *list* of allowed HTTP methods. This list is comma-separated, and the browser must check that the method of the non-simple request is contained within the list of allowed methods. The server sends a list of methods since this will be useful in caching, as we will see shortly. Similarly, and for the same reason, the `Access-Control-Allow-Headers` header contains a list of all non-simple headers supported by the server. The browser must check that the headers (if any) it sent in the `Access-Control-Request-Headers` header are a subset of the headers the server says it supports. For example, in response to the preflight headers on page 21, the server might allow the request by replying with:

- `Access-Control-Allow-Origin: https://browseraudit.com`
- `Access-Control-Allow-Method: GET, POST, PUT, DELETE`
- `Access-Control-Allow-Headers: X-My-Header, X-Another-Header`

Following a successful handshake, the browser will send the actual non-simple request to the server. The request is still sent with the `Origin` header, and the server will respond with `Access-Control-Allow-Origin` just as with a simple request. The browser will check the origin one last time before passing the retrieved data onto the caller. Figure 2.2 shows an overview of a successful non-simple CORS request.

[3]A request header is a non-simple header if it is does not match the list of headers on page 20

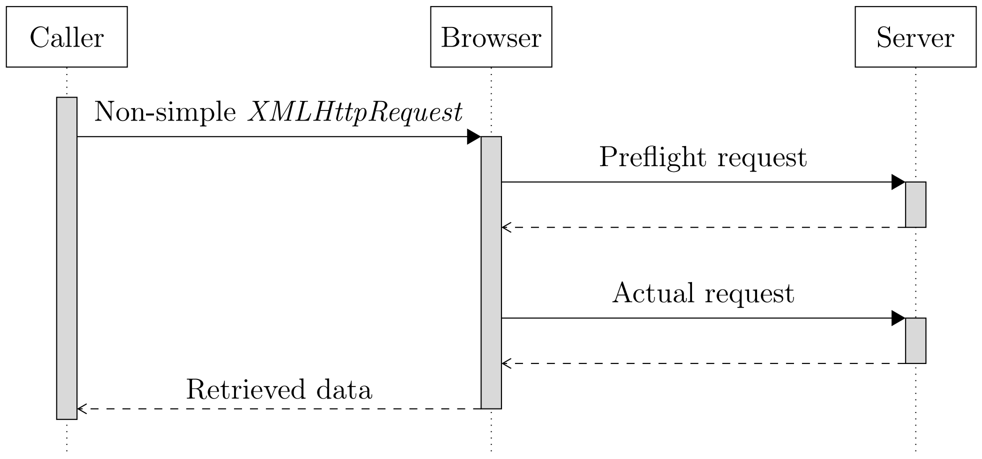


Figure 2.2: A high-level overview of a successful non-simple request with CORS

Caching

The result of a preflight check may be cached by the browser for performance reasons, since every non-simple request using CORS would otherwise require two HTTP requests. The server can specify the maximum age of this cache with the `Access-Control-Max-Age` response header, after which the browser must send a new preflight request. This explains why, in the server's response to a preflight request, the `Access-Control-Allow-Method` and `Access-Control-Request-Headers` headers contain lists of allowed values rather than just repeating the values sent by the browser. Using lists makes the preflight responses as generic as possible so that a single cached response will apply to multiple different requests made by the browser. This avoids the need for another preflight request until the cache expires.

Requests with Credentials

By default, cookies are not included with CORS requests. The Boolean `withCredentials` property of an *XMLHttpRequest* can be used to enable cookies. For this to work, the server's response must include the header `Access-Control-Allow-Credentials: true`. The browser should reject any request, and not pass its data back to the caller, if `withCredentials` is set to `true` but the server's response does not contain the correct header. When the server is responding to a credentialed request, it must specify an origin in the `Access-Control-Allow-Origin` header and cannot use the wildcard value [21].

Once credentials are enabled, the browser will send any cookies it has for the remote domain with the CORS request. The response from the server can also *set* cookies for the remote domain. Note that the same-origin policy still applies to the cookies – the

JavaScript making the CORS request should not be able to view the cookies belonging to the remote domain just because they will be included in the request and returned with the response.

Access-Control-Expose-Headers

The `Access-Control-Expose-Headers` header can be set by the server to specify additional response headers to which the *XMLHttpRequest* object should have access. The *XMLHttpRequest* object has a `getResponseHeader()` method that can be used by the caller to access a given response header from the CORS response. By default, this method can only be used to access the simple response headers[4]. In order to enable the caller to read other headers, the server must use the `Access-Control-Expose-Headers` header.

Rationale for the Split

It is reasonable to question the rationale for separating CORS requests into simple and non-simple requests and handling each type differently. One might wonder why not all requests are treated the same. There is no comment on this in the Design Decision FAQ section of the CORS specification. We found two different explanations for this whilst researching the background on Cross-Origin Resource Sharing.

In his book *The Tangled Web*, Michal Zalewski states that the reason for the split comes down to a trick used in the past by some websites [42]. Before CORS, it was not possible for an attacker to insert custom request headers into a cross-origin *XMLHttpRequest*. Websites could therefore use the presence of some custom request header as proof that the request originated from the same origin as the destination. With the introduction of CORS, this no longer held true. It was originally the plan for almost all CORS requests to be passed straight to the server (like simple requests are today), but the specification had to be revised due to the problems caused by websites that used custom headers as a proof of *XMLHttpRequest* origin. The problem was fixed by the introduction of the two-step handshake.

The *HTML5 Rocks* tutorial on CORS states that "simple requests are characterised as such because they can already be made from a browser without using CORS" [20]. The author argues that an attacker could create simple CORS requests through other scripting methods if he was able to inject his own client-side script. For example, a POST request could be made by creating a form and then submitting it.

[4]The simple response headers are `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified` and `Pragma`

Internet Explorer's Competitor: XDomainRequest

CORS as we have described it (i.e. an extension to *XMLHttpRequest*) is supported today in Internet Explorer versions 10 and 11. Prior to this, in versions 8 and 9, Microsoft implemented only a counterproposal to CORS. This counterproposal is *XDomainRequest*, an API similar to *XMLHttpRequest* but with some key differences when compared to the CORS specification. Cross-origin requests with *XDomainRequest* are much simpler than those with CORS: no custom HTTP headers or methods can be used, and there is no support for credentials. Microsoft's proposal never became popular – the W3C backed the CORS specification and so there was no motivation for any other browsers to implement *XDomainRequest*.

2.2.4 Cookies

When discussing the basics of HTTP cookies in Section 2.1.1, we mentioned two cookie attributes *HttpOnly* and *Secure* that are browser security features. These are both flags rather than attributes – they have no values. The presence of each merely defines a behaviour that the browser should follow when handling the cookie.

HttpOnly

The *HttpOnly* flag instructs the browser to reveal that cookie only through an HTTP API. That is, the cookie may be transmitted to a server with an HTTP(S) request, but should not be made available to client-side scripts. This means that *HttpOnly* cookies cannot be read in JavaScript through the `document.cookie` property like normal cookies can be. The security benefit of this is that, even if a cross-site scripting vulnerability is exploited, the cookie cannot be stolen. Just like the Content Security Policy, the *HttpOnly* cookie flag provides defence in depth.

HttpOnly cookies are supported by all major browsers. The only notable exception is Android 2.3's stock browser.

Secure

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. is 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. In contrast, if the cookies were transmitted in plaintext HTTP requests and the attacker was observing the connection between the victim and the server, the attacker would be able to steal the cookies. This is what the *Secure* flag was designed to prevent, protecting sensitive cookies by ensuring they are only ever transmitted over a secure transport.

The *Secure* flag is supported by all major browsers.

2.2.5 HTTP Strict Transport Security

When discussing the motivations for HTTP Secure in Section 2.1.2, we briefly mentioned man-in-the-middle (MITM) attacks. These attacks take place when an attacker is able to intercept the traffic between a victim's browser and the server. When successfully exploited, a MITM attack can cause a lot of damage. For example, over an unencrypted connection, the attacker is able to view the requests and responses (including cookies and any private data in the server responses). He can also modify the communications, for example he may modify the server's response to inject some of his own content or malicious JavaScript code.

All of these attacks are only possible when the client/server communication is neither encrypted nor authenticated. Over a secure (HTTPS) connection, both requests and responses are encrypted and authenticated. This means that the attacker cannot read any plaintext data like he could earlier, nor can he modify any requests or responses. Assuming that the target server's private key has not been compromised, the only case in which a MITM attack can be successful over HTTPS is if the client (browser) does not check the security certificate used. In this case, the attacker can generate his own (invalid) security certificate, pretending to be the server, and pass this to the victim's browser. This should result in a warning in all web browsers.

HTTP Strict Transport Security (HSTS) is a security mechanism that allows a server to instruct browsers to communicate with it only over a secure (HTTPS) connection for that domain. It exists mainly to defend against man-in-the-middle attacks as described above. The server sends this instruction with a `Strict-Transport-Security` response header, as defined in RFC 6797 [18]. The header is required to include a `max-age` directive, specifying the number of seconds for which the browser should follow the HSTS policy. There is also an optional `includeSubDomains` directive, specifying that the HSTS policy should apply to subdomains of the current host as well as the host itself. Example HSTS headers might be:

- `Strict-Transport-Security: max-age=300`
- `Strict-Transport-Security: max-age=3600; includeSubDomains`

When HSTS is enabled on a domain, the browser must rewrite any plain HTTP requests to that domain to use HTTPS. This includes both URLs entered in the navigation bar by the user and elements loaded by a webpage.

HSTS also instructs the browser to terminate any secure transport attempts upon any and all secure transport errors and warnings. This means that if there is any problem at all with the secure transport, the browser should terminate the request. This applies also to websites using self-signed certificates or certificates whose CA is not embedded in the browser or operating system. Once HSTS is enabled in these scenarios, the browser should refuse to communicate with the server. This applies even if the user has manually whitelisted the certificate in his browser. In other words, when a browser would normally display a bypassable warning regarding secure transport to the user, under HSTS it must instead immediately terminate the request.

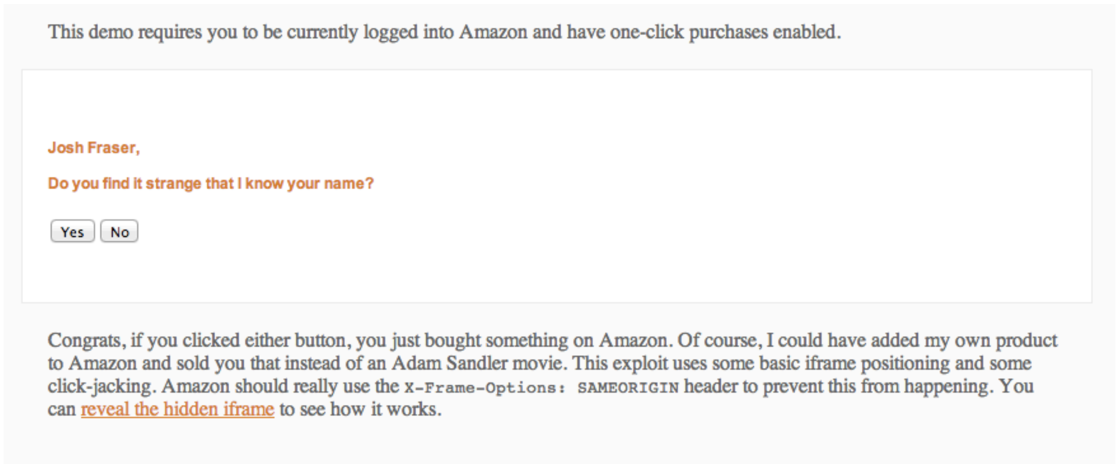
The `Strict-Transport-Security` header should only be sent in a response sent over HTTPS. If a browser receives the header in a response sent over plain HTTP, it should be ignored.

2.2.6 Clickjacking and X-Frame-Options

Clickjacking, also known as UI redressing, is a type of attack in which a user is tricked into clicking on something other than what he believes he is clicking on. The attacker is "hijacking" clicks intended for one application and instead routing them to some other page. Clickjacking can be seen as an instance of the confused deputy problem. It is a very serious attack, since it can be used to take advantage of the fact that a typical web user is often logged into his personal user account on many web applications at any given time. If an attacker is able to hijack a user's clicks, the attacker could trick the innocent user into carrying out a privileged action on his account on a website.

One real-world example of clickjacking is an attack in 2009 that tricked Twitter users into sending a tweet without realising [37, 34]. The attack took the form of a textual link that said "don't click". When logged-in Twitter users clicked the link, they inadvertently sent out a tweet that reposted the link for other people to click on. This attack worked by placing a transparent frame on top of the textual link. In this instance, the attack was fairly harmless, but could be perhaps extended into tricking users into "following" another account on Twitter. This is something that many individuals and companies are interested in doing (and can pay for as a service!) to boost their social media presence.

Pictured in Figure 2.3 is a more serious real-world example of clickjacking. It is a proof-of-concept attack for online retailer Amazon that was recently disclosed [16]. The attack shows that a user could easily be tricked into purchasing any product on Amazon, provided he is logged in and has one-click purchasing enabled. As with the Twitter


This demo requires you to be currently logged into Amazon and have one-click purchases enabled.

Josh Fraser,

Do you find it strange that I know your name?

Congrats, if you clicked either button, you just bought something on Amazon. Of course, I could have added my own product to Amazon and sold you that instead of an Adam Sandler movie. This exploit uses some basic iframe positioning and some click-jacking. Amazon should really use the `X-Frame-Options: SAMEORIGIN` header to prevent this from happening. You can reveal the hidden iframe to see how it works.

Figure 2.3: A proof-of-concept clickjacking exploit on Amazon

clickjacking attack, it works by cleverly positioning a transparent frame containing an Amazon product page, tricking the user into clicking the “Buy Now” button. The exploit has since been fixed by Amazon, using the browser security feature we are about to describe.

Clickjacking is very much a browser vulnerability, and is one that is difficult to prevent. `X-Frame-Options` is a server-side technique that aims to prevent clickjacking. It is supported in all modern browsers, namely Internet Explorer 8+, Firefox 3.6.9+, Opera 10.5+, Safari 4+ and Chrome 4.1+. Its implementation in current browsers is documented in RFC 7034 [32].

`X-Frame-Options` is an HTTP response header that specifies whether or not the document being served is allowed to be rendered in a `<frame>`, `<iframe>` or `<object>`. More specifically, the header specifies an *origin* that is allowed to render the document in a frame. The header must have exactly one of three values:

- `DENY`
- `SAMEORIGIN`
- `ALLOW-FROM` *src*

`DENY` states that a browser must not display the content in any frame. `SAMEORIGIN` states that a browser must not display the content in a frame from a page of different origin than the content itself. `ALLOW-FROM` states that a browser must not display the content in a frame from a page of different origin to the specified source. Only one origin may be stated in the `ALLOW-FROM` value; there is no wildcard value. Some example

`X-Frame-Options` headers are as follows:

- `X-Frame-Options: DENY`
- `X-Frame-Options: SAMEORIGIN`
- `X-Frame-Options: ALLOW-FROM https://test.browseraudit.com/`

Not all browsers supporting the `X-Frame-Options` header support the `ALLOW-FROM` value type. It is also important to note that today's browsers implement differing behaviour when it comes to nested frames. Consider the case where a page A loads a page B inside a frame, which in turn loads a page C inside a frame. Page C is served with an `X-Frame-Options` header restricting the origin that is allowed to contain it within a frame. In this scenario, some browsers will compare the origin of page C to page A (the top-level browsing context) whereas other browsers will compare the origin of page C to page B. This limits the number of different cases we can test, which we describe in more detail when discussing the implementation of our tests for this feature in Section 5.6.1.

When an `X-Frame-Options` policy is violated, i.e. an origin mismatch occurs when framing a document that is served with the `X-Frame-Options` header, different browsers exhibit different behaviour. Some browsers will present the user a message that allows him to open the document in a new window, so that it becomes clear that it is a separate document and there is no clickjacking threat. Other browsers are less forgiving and simply render an empty document.

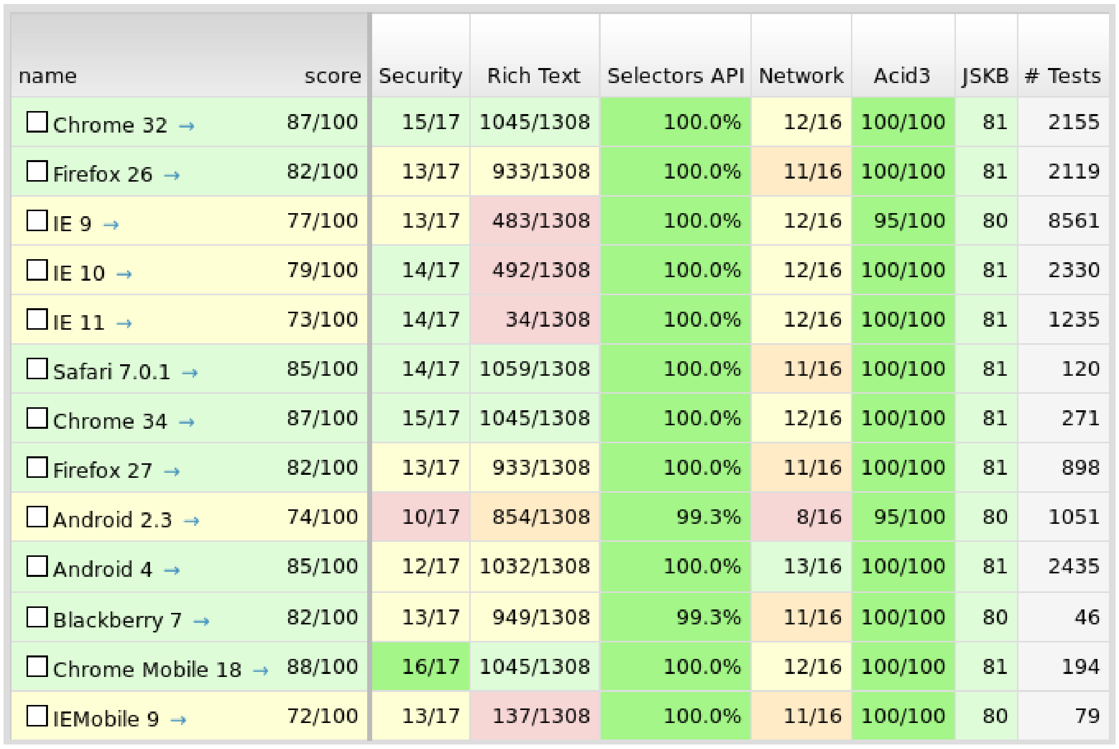
`X-Frame-Options` must be sent as an HTTP header field and should be explicitly ignored by browsers when declared with a meta `http-equiv` tag.

2.3 Related Work

In this section we discuss related work found during the initial research for this project. There are some projects that have similarities to ours, but none of them tries to achieve what we believe to have achieved: a comprehensive report of the user's browser security, produced purely by accessing a single webpage with minimal interaction required from the user. We consider the offerings of some of the related work, and what we learned from them with respect to our project.

2.3.1 Browserscope

Perhaps the most similar work to our project is *Browserscope*[5]. Describing itself as a "community-driven project for profiling web browsers", the project homepage detects which browser you are using before inviting you to run their tests on your browser. These tests cover a wide range of browser features such as network performance, CSS selectors supported and, most interestingly, security. Once the tests are complete, the browser version tested and its results are added to the project's database. These results are then aggregated and made publicly available, making it easy to keep track of functionality across all browsers that have been tested. *Browserscope* aims to be a useful tool for web developers, allowing them to determine which modern browser features are widespread enough that they are safe to use in a popular web application. Figure 2.4 shows a summary of their test results for today's most popular browsers.



name	score	Security	Rich Text	Selectors API	Network	Acid3	JSKB	# Tests
☐ Chrome 32 →	87/100	15/17	1045/1308	100.0%	12/16	100/100	81	2155
☐ Firefox 26 →	82/100	13/17	933/1308	100.0%	11/16	100/100	81	2119
☐ IE 9 →	77/100	13/17	483/1308	100.0%	12/16	95/100	80	8561
☐ IE 10 →	79/100	14/17	492/1308	100.0%	12/16	100/100	81	2330
☐ IE 11 →	73/100	14/17	34/1308	100.0%	12/16	100/100	81	1235
☐ Safari 7.0.1 →	85/100	14/17	1059/1308	100.0%	11/16	100/100	81	120
☐ Chrome 34 →	87/100	15/17	1045/1308	100.0%	12/16	100/100	81	271
☐ Firefox 27 →	82/100	13/17	933/1308	100.0%	11/16	100/100	81	898
☐ Android 2.3 →	74/100	10/17	854/1308	99.3%	8/16	95/100	80	1051
☐ Android 4 →	85/100	12/17	1032/1308	100.0%	13/16	100/100	81	2435
☐ Blackberry 7 →	82/100	13/17	949/1308	99.3%	11/16	100/100	80	46
☐ Chrome Mobile 18 →	88/100	16/17	1045/1308	100.0%	12/16	100/100	81	194
☐ IEMobile 9 →	72/100	13/17	137/1308	100.0%	11/16	100/100	80	79

Figure 2.4: *Browserscope*'s summary of the current top browsers

The most relevant aspect of *Browserscope* to our project is the security tests: *Browserscope* currently runs 17 browser security tests as part of its testing. These tests run automatically in the browser in a similar fashion to how our tests will run. The *Browserscope* security tests provided us with some good initial test ideas for BrowserAudit. All of their security tests cover standard security features of a browser, which one would

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

hope are correctly implemented in each of today's major browsers. The list of security features covered by *Browserscope* is far from exclusive, however: there are many more standard security features that we should also be able to test.

When the *Browserscope* tests are run, the output is simply a list of which tests passed and which tests failed. There is no indication of why each test passed or failed, and no real technical explanation of which browser security feature the test is trying to break. This is perfectly fine for a browser profiling project, however in our project we want to provide a much more thorough and technical breakdown of each test result. Which security feature does the test cover? How is it being tested? What is the expected result? What was the actual result? For browser developers and security researchers, all of this information will be essential.

2.3.2 How's My SSL?

How's My SSL?[6] is a young project that tells the user how secure their TLS client[7] (probably a web browser) is. It is so young, in fact, that it was launched after the initial research for our project began.

The *How's My SSL?* application 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 tells the server it can support. The site then performs a security assessment on the TLS client and reports the results in a very clear manner, with "Learn More" links for more technical background. This layout works well, and we took ideas from this for BrowserAudit. We also realised that we could potentially make use of *How's My SSL?* within our project, since the security of a browser as a TLS client is very relevant to the security of the browser itself. A JSON API is provided, whose response includes all of the information on the verdict on the TLS client's security. These results could be presented to the user alongside the results of our own tests.

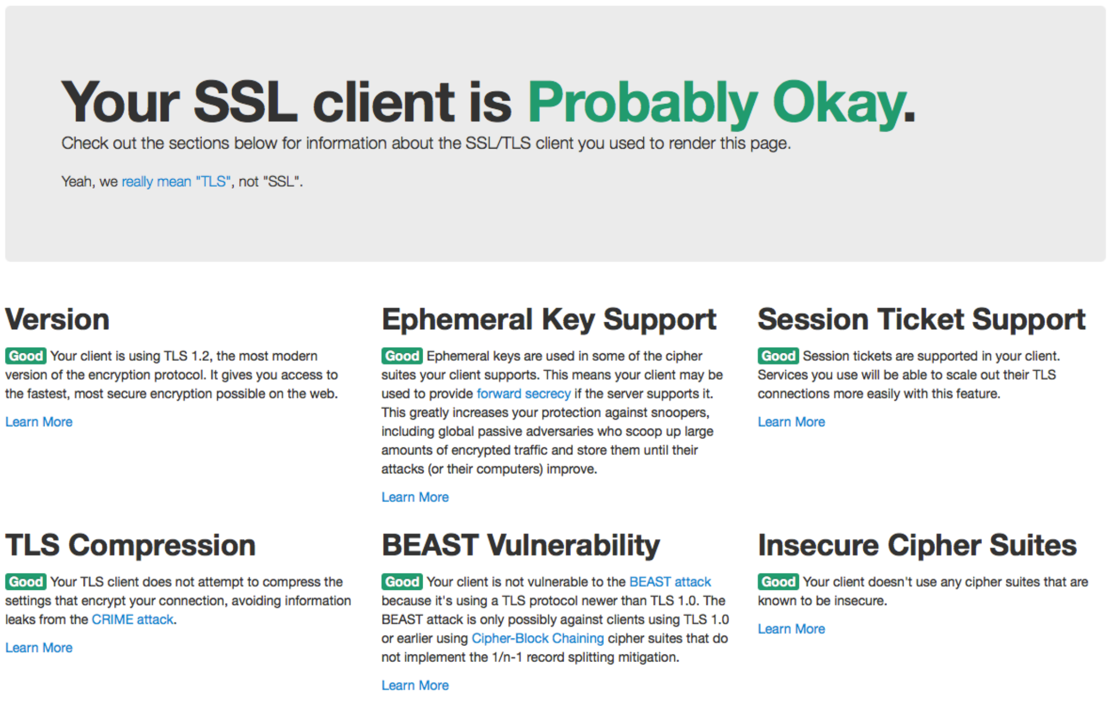
2.3.3 Panopticlick

Panopticlick[8] is an experiment run by the Electronic Frontier Foundation to investigate how unique – and therefore trackable – modern web browsers are, by fingerprinting the version and configuration information that a browser transmits to websites with its requests. Some of the information fingerprinted by *Panopticlick* does not come directly

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

[7]TLS stands for Transport Layer Security. It is the most commonly used protocol for encrypting data across the Internet, and is used by the HTTPS protocol to load webpages securely. We covered this in more detail in Section 2.1.2

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



Your SSL client is Probably Okay.

Check out the sections below for information about the SSL/TLS client you used to render this page.

Yeah, we really mean "TLS", not "SSL".

Version

Good Your client is using TLS 1.2, the most modern version of the encryption protocol. It gives you access to the fastest, most secure encryption possible on the web.

Learn More

Ephemeral Key Support

Good Ephemeral keys are used in some of the cipher suites your client supports. This means your client may be used to provide forward secrecy if the server supports it. This greatly increases your protection against snoopers, including global passive adversaries who scoop up large amounts of encrypted traffic and store them until their attacks (or their computers) improve.

Learn More

Session Ticket Support

Good Session tickets are supported in your client. Services you use will be able to scale out their TLS connections more easily with this feature.

Learn More

TLS Compression

Good Your TLS client does not attempt to compress the settings that encrypt your connection, avoiding information leaks from the CRIME attack.

Learn More

BEAST Vulnerability

Good Your client is not vulnerable to the BEAST attack because it's using a TLS protocol newer than TLS 1.0. The BEAST attack is only possibly against clients using TLS 1.0 or earlier using Cipher-Block Chaining cipher suites that do not implement the 1/n-1 record splitting mitigation.

Learn More

Insecure Cipher Suites

Good Your client doesn't use any cipher suites that are known to be insecure.

Learn More

Figure 2.5: *How's My SSL?*'s verdict on Firefox 27.0. This is an improvement on the **Bad** rating given to Firefox 26, which was a result of two issues: lack of TLS 1.2 support, and support for the possibly-insecure `SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA` cipher suite

from browser requests, but is instead readily available thanks to the presence of JavaScript and browser plugins such as Adobe Flash Player.

Visitors click the "Test Me" button and are then provided with their uniqueness score. For example, running in Firefox 27.0 on a slightly customised Ubuntu 12.04 installation:

> Your browser fingerprint **appears to be unique** among the 3,846,235 tested so far.
>
> Currently, we estimate that your browser has a fingerprint that conveys **at least 21.88 bits of identifying information.**

Below the uniqueness score is a breakdown of all of the measurements used to obtain the result. These measurements include the user agent string, HTTP `Accept` headers, browser plugin details, screen size and colour depth, and system fonts. The data are then anonymously stored in the project database to make future uniqueness scores more accurate, and to allow for analysis of the data. A paper was published in 2010 reporting the statistical results of the experiment. When a browser was chosen at random, it was expected that at best only one in 286,777 browsers would share its fingerprint [14]. This level of uniqueness offers interesting visitor tracking opportunities to those who could gain something from it. For example, advertising companies could track users around the Web, targeting advertisements at them based on their online footprint.

As with *How's My SSL?*, the most interesting aspect of *Panopticlick* from our project's perspective is the user experience. It is very simple for the user to run *Panopticlick*'s test on their browser – a single click is all that's required. We believe that this click is important too, since it explains to the user what is about to happen before their tests are actually executed. Once the testing is complete, a very simple uniqueness summary is provided, with the technical information lower down on the page. This means that the test results appeal to a wide audience – a more typical web user can visit the page to receive a rough assessment on how easily he can be tracked, while a more advanced user can also study the exact information his browser is transmitting to each website he visits. He may then modify his browser settings in order to make himself less trackable around the Web.

2.3.4 BrowserSpy

BrowserSpy[9] describes itself as "the place where you can see just how much information your browser reveals about you and your system". *BrowserSpy* currently offers 75 tests that can be run individually – there is no option to run all tests automatically since the output of each test is rather verbose. Each of the tests aims to show the user how much information can be retrieved from their browser just as a result of visiting a single test page. Just like *Panopticlick*, some of these tests are based on the information the browser sends to the server with its requests, whereas others are more complicated and make use of client-side scripting. Another comparison to *Panopticlick* is that *BrowserSpy*'s main focus is privacy, as opposed to security which is our main focus. That said, some of *BrowserSpy*'s tests are security-based, however they are not presented in this manner. For example, Table 2.2 shows an example output of *BrowserSpy*'s cookies test. We can see that some of this output is indeed related to browser security, namely the *HttpOnly* cookie attribute that we described in Section 2.2.4. Note that the `browserspy_server_httponly` cookie set by the server-side cannot be read by the JavaScript client-side: this is correct behaviour, as highlighted by the "Can JavaScript read *HttpOnly* Cookies?" result. Another security-related test offered by *BrowserSpy* tries to exploit CSS to determine the user's browsing history.

BrowserSpy has a similar architecture to our project in that it runs tests (some of which are testing security features) on a users browser. The client- and server-side of the application work together to run the tests, in the same way that we will have our server-side utilising as many browser security features as possible such that the client-side can

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

Test	Result
Cookies enabled?	Yes
JavaScript cookies supported?	Yes
Server cookies supported?	Yes
HttpOnly cookies supported?	Yes
Can JavaScript read *HttpOnly* cookies?	No – which is correct
Meta tag cookies supported?	Yes
Number of cookies?	3
Current cookies	`popunder=yes; popundr=yes; setover18=1`
Current cookies by server-side	`browserspy_server_httponly=test`
Max no. of JavaScript cookies per server	100
Max size per cookie	4,000 bytes

Table 2.2: An example output of *BrowserSpy*'s cookies test

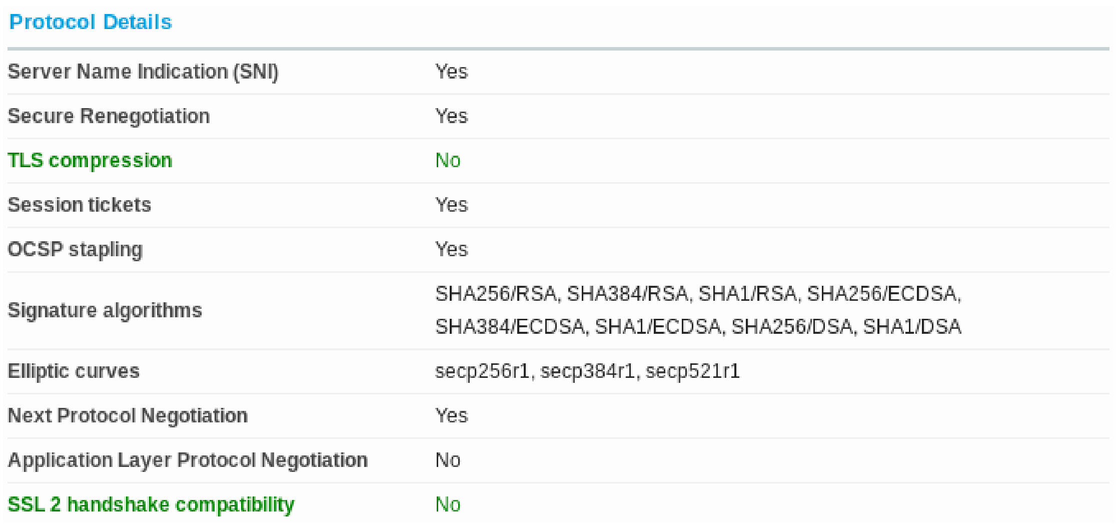
then test their implementations. In *BrowserSpy*, the server-side is written in PHP, whilst the client-side is mostly JavaScript but also makes use of VBScript, Java and Flash when available.

The level of detail provided in the *BrowserSpy* test outputs is an improvement on *Browserscope* however, for the purposes of our project, we still believe that it will be useful to also provide implementation details of each test for a more technical audience. We also learned from *BrowserSpy* that it is frustrating having to run one test at a time, and so we will prefer an interface similar to those of *Panopticlick* and *How's My SSL?* for BrowserAudit.

2.3.5 Qualys SSL Labs

Qualys is a provider of cloud security. They also have a non-commercial research website, *SSL Labs*[10], that they describe as a "collection of documents, tools and thoughts related to SSL". One of these tools is very popular in the web development world and used by many webmasters to test a server's SSL configuration. This involves checking the cipher suites it supports and the certificate it presents. Simply enabling SSL on a server and presenting a valid certificate is not enough for maximum security. Many default SSL configurations will score poorly in the *SSL Labs* server test, and so many webmasters use it to fine-tune their TLS configuration. In October 2013, an SSL *client* test was added to the *SSL Labs* website. This is of more interest to us, since browsers behave as an SSL/TLS client whenever they load a page over HTTPS. These tests, loaded on a single page, automatically assess the user's browser as a TLS client.

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



Protocol Details

Server Name Indication (SNI)	Yes
Secure Renegotiation	Yes
TLS compression	No
Session tickets	Yes
OCSP stapling	Yes
Signature algorithms	SHA256/RSA, SHA384/RSA, SHA1/RSA, SHA256/ECDSA, SHA384/ECDSA, SHA1/ECDSA, SHA256/DSA, SHA1/DSA
Elliptic curves	secp256r1, secp384r1, secp521r1
Next Protocol Negotiation	Yes
Application Layer Protocol Negotiation	No
SSL 2 handshake compatibility	No

Figure 2.6: A section of example output from the *SSL Labs* client test

The *SSL Labs* client test is similar to the *How's My SSL?* project that we have seen already, although the output is perhaps more useful. The side-effect of this is that the *SSL Labs* test output is also much more complicated. The output very much appears to be targeted at highly technical users only – there is very little text describing what each test is doing. In fact, the only real indicator at all is the headings at the top of each batch of tests. Figure 2.6 shows a screenshot of some example output from the *SSL Labs* client test. We can see a breakdown of the TLS capabilities of our browser (Firefox 29.0). Other than the green text in two rows indicating a good result, a non-expert user might have difficulties understanding what the results mean.

Looking towards how this project relates to BrowserAudit, we learned that technical output is very helpful but should perhaps be supplemented with a clearer summary (as we saw in *How's My SSL?*) for users who may not know much about the inner details of the browser features being tested.

2.3.6 CanIUse Test Suite

CanIUse[11] is a website referenced often in this report when discussing browser support for various security features. It offers compatibility data for a wide variety of browser features beyond security, such as support for HTML5 and CSS3 features. For each feature, it displays a table detailing support for the feature in both desktop and mobile browsers. This table highlights the versions (both current, past and future) of each browser in which the feature became supported or is planned to be supported in the

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

future. *CanIUse* sometimes also has notes on quirks in the implementation of a feature in a particular browser.

A lesser-known public feature of *CanIUse* is their test suite[12]. This is the test suite that they use when testing support for various web technologies to display in their browser support tables. The tests for all features are executed on page load, but it is also possible to click a link for each feature to test only that feature. There are four different types of test:

- **auto** – automated JavaScript tests;
- **visual** – requires visual comparison to confirm;
- **visual-square** – also requires visual confirmation from the user; test must create a green square;
- **interactive** – requires interaction to confirm support.

Having these different types of test makes perfect sense. The automated JavaScript tests are preferred when possible, but many of the tests require some sort of human confirmation since they are testing things like CSS3 page style features. In some, it is just a case of looking for a green square, whilst others require a comparison to an image provided of the expected output. We stated in our Introduction chapter that we aim to automatically test browser security features with BrowserAudit, and we have so far been able to avoid any kind of visual confirmation being required from the user when writing our own tests. It is interesting to note, however, that the visual confirmation works well on the *CanIUse* test page, should we encounter a situation in the future in which we believe a security test is important but the feature cannot be automatically tested.

Content Security Policy Test

There are not many security tests in the *CanIUse* test suite, however there is one that we found to be especially interesting: Content Security Policy. This is a single test that tests a browser's support for the Content Security Policy (CSP). This is a feature that BrowserAudit tests today. Interestingly, the *CanIUse* test for CSP support requires a visual confirmation, whilst we have been able to test it automatically in JavaScript. The CSP is also a very detailed security feature (see Section 2.2.2) that requires multiple tests to cover, rather than a single test as in the *CanIUse* suite. The use of just a single test in the *CanIUse* suite is understandable, since it is intended only to test support for the feature and not its implementation.

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

3 Design

In this chapter we discuss the design of our application. This includes our chosen implementation languages and key libraries used. We showcase BrowserAudit's front-end website design with screenshots, and also describe our server-side system architecture.

BrowserAudit consists of both a client-side application and a server-side application. The client-side runs in the user's browser whilst the server-side runs on a web server. The two parts of our application work together in order to run our security tests on the user's browser. Put simply, the server-side of the application exercises browser security features and the client-side application then tests that these features are implemented as expected.

Our application is very modular, i.e. browser security tests can be added and removed independently of each other. This means that the application lends itself nicely to an incremental software development workflow – we can write, test and introduce each of our security tests one at a time. The key benefit of this is that the application is always in a demonstrable state, since the introduction of a new test should never break an already-existing test (at this stage this is true; perhaps in some future work it will make sense for BrowserAudit tests to be dependent on and "include" others). The development cycle should be as simple as deciding on the next test to implement, writing it, testing it, and introducing it to the webpage that so that it is run alongside the other tests when the user accesses our web application. As we look towards design decisions, we must take the modular nature of the project into account and ensure that our program design allows for an incremental workflow as described.

3.1 Client-Side

The client-side of our application is written in JavaScript, the only client-side scripting language that is fully supported across all of today's browsers. It is responsible for running the tests and reporting the results to the user. A typical test involves making multiple requests to the server-side application (often through AJAX, although also through other methods such as loading an image) and checking the responses against the expected responses.

Rather than using any large frameworks or syntactic sugar, we instead write our tests in raw JavaScript supplemented only by jQuery to simplify DOM manipulation and AJAX requests. We are considering plans to open source the project, inviting other developers to contribute to our testing codebase by adding their own tests covering new security features. Should we end up doing this, we would not want to limit the number of developers who might be able to contribute to BrowserAudit by using an implementation framework that many developers won't be familiar with. Many modern client-side applications nowadays make use of CoffeeScript, a language that compiles down to JavaScript. It has much nicer syntax than plain JavaScript, borrowing syntactic sugar from languages such as Ruby, Python and Haskell. We opted not to use this because not all JavaScript developers will be comfortable with Coffeescript, and it is important that we don't limit the number of potential contributors to our project.

3.1.1 Mocha

Whilst considering the application design for BrowserAudit, there was a clear duality between our project and *unit testing* – the software testing method whereby the smallest possible units of code are tested individually to ensure they are fit for use. For example, when unit testing a Java application with a framework such as JUnit, one writes his tests in separate source files and runs them either on the command line or, more likely, instructs his IDE to run them all for him and report back on the test results. This is very similar to the usage pattern we wish to achieve: we want the user to visit our page, which will then run our tests and report back on the results.

Keeping in mind the desire for a design allowing tests to be as modular as possible, we looked into JavaScript unit testing frameworks. If we could find an appropriate client-side JavaScript unit testing framework, this would be able to handle the automatic running of tests and reporting of results, allowing us to focus on the implementation of the tests themselves. After researching the various options (many of which were for server-side JavaScript only, targeting the increasingly-popular node.js platform) we settled on **Mocha**[1], which describes itself as a "feature-rich JavaScript test framework running on node.js and the browser, making asynchronous testing simple and fun". Mocha allows us to run unit tests in the browser. The asynchronous property is also highly important to us, since many our tests will involve making asynchronous requests to the server-side of our application.

Listing 4 shows the JavaScript client-side of a proof-of-concept browser security test we wrote to decide whether Mocha is fit for our needs. It tests that the browser correctly implements *HttpOnly* cookies (see Section 2.2.4). The simplicity of this test demonstrates the power of the JavaScript testing library; the test was easy to write and is simple to

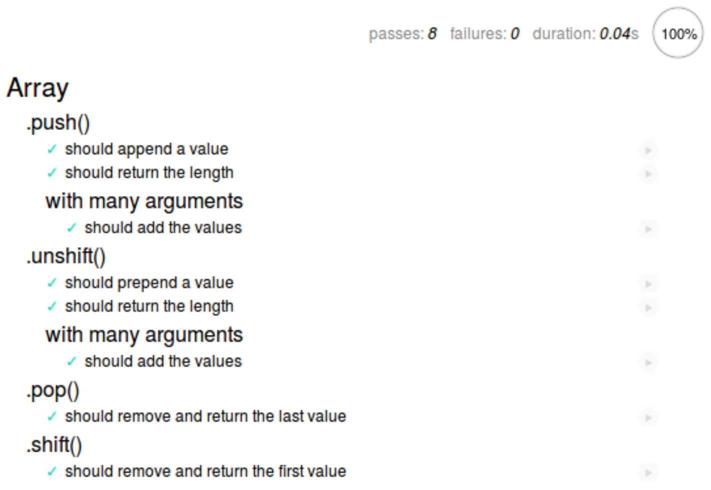
[1] `http://visionmedia.github.io/mocha/`

```
$.get("/del_httponly_cookie", function() {
  expect($.cookie("httpOnlyCookie")).to.be.undefined;
  $.get("/set_httponly_cookie", function() {
    expect($.cookie("httpOnlyCookie")).to.be.undefined;
    done();
  });
});
```

Listing 4: The client-side of a proof-of-concept *HttpOnly* cookie test

understand. Firstly, a call is made to a server-side page to clear any leftover cookie value from a previous test run. After this, we expect its value to be undefined. We then make a call to a server-side page which sets the cookie. Since we're trying to access the cookie from JavaScript, we still expect it to be undefined. If this isn't the case, then the browser has failed the test since it has allowed our script to read the value of an *HttpOnly* cookie. The call to `done()` informs Mocha that the asynchronous test is now complete. We are making use of jQuery here to simplify the requests made to the server and the reading of cookies from JavaScript.

Mocha allows us to use any assertion library when writing tests. We are using Chai[2] with the Expect interface, allowing us to write assertions of the form `expect(foo).to.equal("bar")`.


```
passes: 8  failures: 0  duration: 0.04s  100%

Array
  .push()
    ✓ should append a value
    ✓ should return the length
    with many arguments
      ✓ should add the values
  .unshift()
    ✓ should prepend a value
    ✓ should return the length
    with many arguments
      ✓ should add the values
  .pop()
    ✓ should remove and return the last value
  .shift()
    ✓ should remove and return the first value
```

Figure 3.1: A screenshot of Mocha's default test output

Of course, Mocha was not designed for our purposes. It is designed for use by developers to debug their code behind the scenes, not for use by an end-user in a front-end application. As such, Mocha's default output – displayed in Figure 3.1 – is not fit for our purposes. Some aspects of it are good, for example the way in which it lists the test

[2] `http://chaijs.com/`

outputs (neatly organised into categories) with tick and cross symbols. Some aspects of it are not appropriate, such as the way in which it flags test runs that take too long to run. This isn't useful in our case since our asynchronous requests to the server-side can take variable amounts of time. We will need to modify Mocha's output styling to make it fit for our needs, taking into account what we learned from the related work in our background research. Fortunately Mocha makes this relatively simple, by allowing developers to write their own result "reporters". When writing a reporter, there are call-backs for events such as the completion of a test that can be used to update a graphical in-browser display.

3.1.2 Website Design

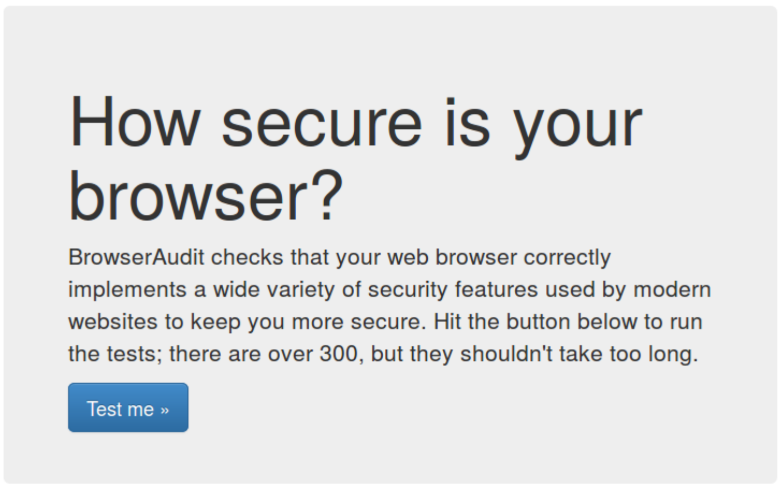
An important aspect to the project is the design of the website itself. After all, this is how our project will be accessed and used. When designing the website we wanted the design to be appropriate for as wide an audience as possible. This means that the basics should be very accessible, such as the number of tests that passed and our overall verdict on the user's browser. We don't want a not-so-technical user to be overfaced by anything technical – they should still be able to use our site comfortably. A highly technical user, however, such as a security researcher or even a browser developer, should be able to access the exact reasons for any test failures: expected results and actual results. For an in-between user, who is technical but not a browser security expert, we should provide some level of technical information about what each category of browser tests is doing. By doing this, BrowserAudit is able to interactively educate users on the importance of browser security by explaining to them the tests that we run on their browsers.

Taking all of this into account, we have produced a design that we believe is suitable for the widest audience possible. It is based heavily around components of Twitter's front-end framework *Bootstrap*[3] which makes it easy to produce a layout that works consistently across all browsers. *Bootstrap* also allowed us to produce a design that did not require us to spend too much time writing and testing our own CSS stylesheets and JavaScript. *Bootstrap* webpages are responsive by default, meaning that they work well even on narrow screens such as mobile devices. This is important to us since browsers on mobile devices can be some of the most infrequently updated and insecure! We will now discuss the main components of our design that we believe contribute towards a positive user experience.

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

Landing Page

When a user visits our website, the tests do not run automatically. This would be confusing for a user if it was his first visit to our site, and could result in him quickly navigating away. Instead, a brief landing page is displayed. This is an idea that we took from the EFF's *Panopticlick* – discussed in Section 2.3.3 – that we think works well. We give the user a quick summary about what is about to happen, and allow him to click a button to start the tests. It is his decision to run our browser security tests; they are not started automatically. Figure 3.2 shows the main call to action on our landing page.



How secure is your browser?

BrowserAudit checks that your web browser correctly implements a wide variety of security features used by modern websites to keep you more secure. Hit the button below to run the tests; there are over 300, but they shouldn't take too long.

Figure 3.2: The call-to-action on our landing page displayed before the tests are run

Test Outcomes: Okay, Warning, Critical

Rather than having the outcome of our security tests being a binary pass or fail, we instead have three possible test outcomes: okay, warning and critical. Our reasoning for this is that some of our tests are testing young browser security features, the absence of which does not necessarily mean that a user should update or replace his browser. We would not want to confuse a user by flagging his browser as insecure for failing such a test. On the other hand, some of our tests are testing features (e.g. the same-origin policy) that absolutely every browser should correctly implement, and so the failure of such a test should result in a more severe warning. Our solution to this is the three possible test outcomes. If a Mocha test passes, it is displayed as okay. If a test fails, it will be displayed as either a warning or critical depending on the test.

Test Summary Box

The main component on the page that actually runs the tests is the summary box displayed at the top of the page. This summary updates in real time as tests complete. An example of how the summary box can look part-way through the execution of our tests is shown in Figure 3.3. There are multiple elements to this that we believe offer a good user experience.

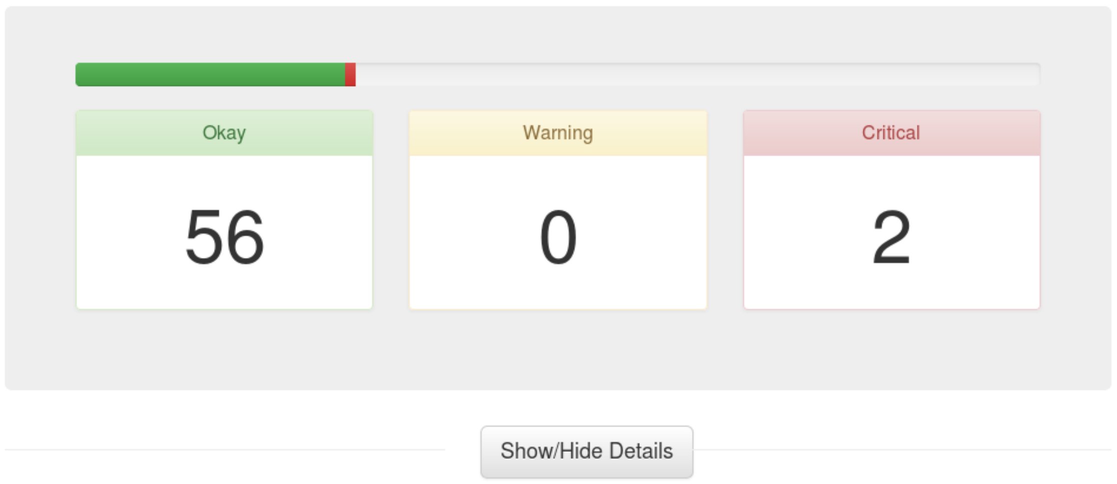


Figure 3.3: The test summary box part-way through the execution of our tests

The most noticeable elements in the summary box are the three boxes containing the current result counts. These very clearly display to the user the number of tests so far that are okay, how many have warnings, and how many are critical. These result counts are augmented by the progress bar above them, which also updates in real time. The progress bar clearly indicates to the user that the tests are still running (since it updates so often) and also gives an accurate indication of how many more tests are still to be run. In addition to this, the progress bar is composed of three colours displaying what proportion of tests so far have each test outcome (okay, warning, critical). This gives a good graphical representation of how the user's browser is fairing as far as our tests are concerned. This output can be easily understood by any user, whatever their technical background.

When all tests are complete, the background of the test summary box changes from grey to be the colour matching BrowserAudit's verdict on the user's browser. This verdict is the worst result encountered during all tests. If all tests are okay, the summary box will turn green. If there are warnings but no critical results, it will be yellow. Otherwise, there are critical failures and so the summary box will turn red.

Show/Hide Details

Below the summary box, also pictured in Figure 3.3, is a button labelled "Show/Hide Details". This can be pressed by more advanced users to learn details about the tests being executed and their results. This is especially interesting if the status box above it is showing any warnings or failures. Figure 3.4 shows how the box displayed after pressing the button may look once all panels have been collapsed. Each of the panels (one for each major test category) is expandable, and any categories with warnings or critical results will be expanded by default. The Content Security Policy category has been manually collapsed (by clicking the category title) in this screenshot.

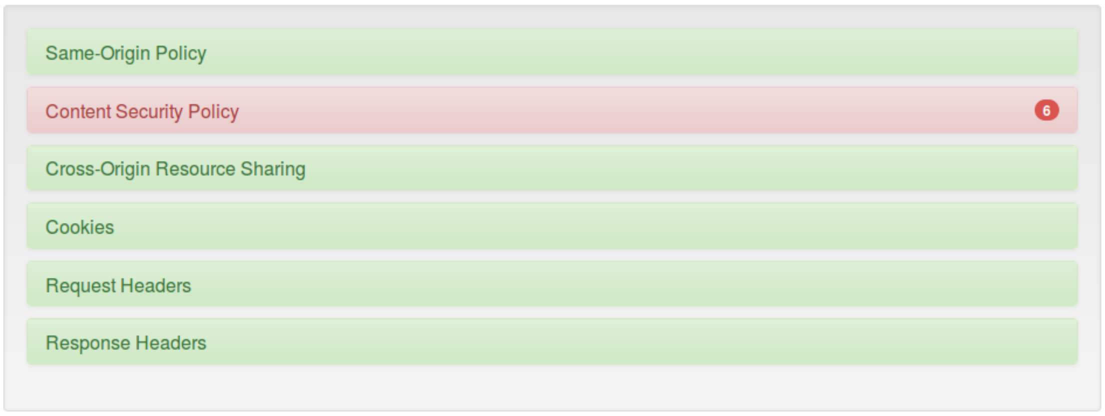
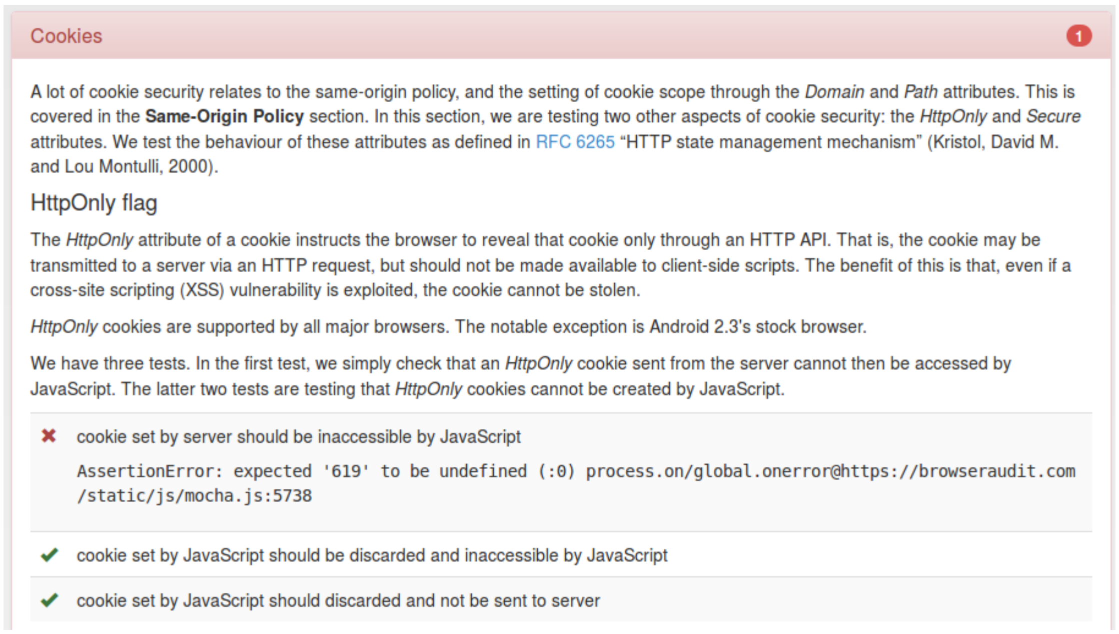


Figure 3.4: The box displayed after "Show Details" is pressed, with all panels collapsed

We can see that there is a panel for each major test category. The background colours of the panel headings reflect the status of the panel, i.e. the colour representing the worst result encountered in the category, in a similar fashion to the main summary box on completion as described already. These background colours are updated live, should a user choose to view the additional details whilst tests are still running. An animated "in progress" icon, not pictured, is displayed alongside the category (if any) whose tests are still in progress. To the right-hand side of the Content Security Policy bar we can also see a numerical "badge" indicating the number of critical tests. A similar badge is also displayed in yellow for any warnings that have occurred.

Expanding a Category Panel

Figure 3.5 shows a screenshot of an expanded category panel. We can see how the category has a paragraph of text explaining what the category of tests is doing. This explanation is targeted at a technical user who may not be well versed in browser security. We can also see an example of a subcategory, "HttpOnly Flag". Each of these subcategories may also have description text, as shown in this case.


Cookies

1

A lot of cookie security relates to the same-origin policy, and the setting of cookie scope through the *Domain* and *Path* attributes. This is covered in the **Same-Origin Policy** section. In this section, we are testing two other 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" (Kristol, David M. and Lou Montulli, 2000).

HttpOnly flag

The *HttpOnly* attribute of a cookie instructs the browser to reveal that cookie only through an HTTP API. That is, the cookie may be transmitted to a server via an HTTP request, but 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. The notable exception is Android 2.3's stock browser.

We have three tests. In the first test, we simply check that an *HttpOnly* cookie sent from the server cannot then be accessed by JavaScript. The latter two tests are testing that *HttpOnly* cookies cannot be created by JavaScript.

✖ cookie set by server should be inaccessible by JavaScript

```
AssertionError: expected '619' to be undefined (:0) process.on/global.onerror@https://browseraudit.com
/static/js/mocha.js:5738
```

✓ cookie set by JavaScript should be discarded and inaccessible by JavaScript

✓ cookie set by JavaScript should discarded and not be sent to server

Figure 3.5: An expanded category panel

The screenshot in Figure 3.5 also shows how each individual test result is displayed. We can see a table with rows of alternating background colours, making it easy to read results across. Tick and cross icons are shown, clearly indicating an okay result and a critical result. In the event of a warning result (not shown), a yellow warning sign is displayed. In the event of a critical test (and also a warning result), we can see the technical explanation displayed below the test title. In this particular test, we can see that an *HttpOnly* cookie set by the server can wrongly be accessed by JavaScript. The error message makes it clear that the test was expecting the cookie's value to be undefined, but it was actually "619". This level of detail is intended to be useful for advanced BrowserAudit users such as browser developers and security researchers.

JavaScript Disabled Message

In the case where a user has JavaScript disabled in his browser, we use a `<noscript>` tag to display a message explaining that he should enable it in order to run our tests. In this message, we acknowledge that having JavaScript disabled by default is indeed a security benefit. We also point out, however, that it is infeasible to automatically test browser security features without it. Some of our tests could in theory work by requesting that the user respond to several stimuli on a page, clicking buttons such as "yes, I see the green box", but this would result in a completely different user experience to what we aim to achieve. There are also some tests that are completely impossible to have without

JavaScript, such as testing that an *HttpOnly* cookie created by JavaScript is immediately thrown away and never sent to a server.

3.2 Server-Side

In this section we describe the design decisions made for the server-side of BrowserAudit. The server-side must handle all of the requests made by the client-side. We first discuss our choice of implementation language, and in Section 3.2.2 we describe our system architecture.

3.2.1 Go

The server-side of our application is written in Go. Otherwise known as golang, Go is a relatively young programming language that has gained significant traction in recent years. It was initially developed at Google in 2007 and open-sourced in 2009 [25]. Go is used in many confidential Google projects, and is also used in backends at many technology companies including BBC Worldwide [1], GitHub [29], Heroku [24], Tumblr [27] and Soundcloud [7].

The three most commonly cited reasons for using Go are its concurrency primitives, ease of deployment, and performance. Only one of these is especially relevant to us: performance. Go is fast. This is important for BrowserAudit because a single user running our browser tests results in hundreds of requests spread over a minute or two. In order to comfortably support multiple concurrent users, we require an implementation language that offers good performance. The other two reasons, concurrency primitives and ease of deployment, are not as relevant to our project. We are not making use of Go's currency primitives ourselves, although it is worth noting that the `net/http` package that we use makes extensive use of golang's currency features, allowing it to efficiently handle multiple concurrent HTTP requests. Ease of deployment may become important to us in terms of scalability, discussed further below.

We also chose Go because of its simplicity. We do not require a web framework that ships with hundreds of files and its own custom build system. Our server-side can and should be very simple. As we will see shortly, its only tasks are session management and serving different HTTP response headers that activate various browser security features. Any additional features could result in unnecessary overhead. Go lets us write our own custom web server from the ground up, resulting in a minimalistic server-side codebase that is easy for another developer to understand. Go is also statically-typed; this means that very few runtime errors occur, since most mistakes are caught at compile-time.

3.2.2 System Architecture

Our web application is currently running on a single Linode[4] cloud server. The server has 2GB of RAM, 2 CPU cores and 48GB of storage, which is plenty for our needs. We chose to host the application on an external server rather than a departmental "DoC Private Cloud" server due to the restrictions imposed by the department on the Cloudstack servers. For example, the standard web ports of 80 and 443 are not publicly accessible on a Cloudstack server – they are only accessible from within the Imperial network. Another reason for wanting to host the application externally is that we hope to continue maintaining the application beyond the timescale of this undergraduate project.

The server runs Ubuntu 14.04 LTS. We chose to use Ubuntu because we have lots of experience with it already (especially running it as a web server) and because its official repositories contain all of the packages we need.

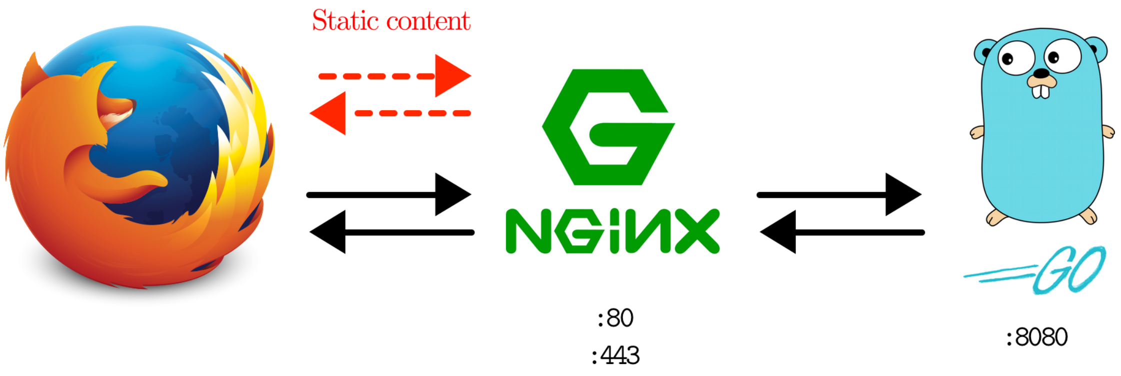


Figure 3.6: Nginx as a reverse proxy in front of our Go web server

Our web application makes use of two web servers: a public-facing Nginx web server running on ports 80 and 443, and a local golang web server running on port 8080. The Nginx server is running as a *reverse proxy* in front of the golang web server. The golang web server is not publicly accessible; all outside requests made to our application are to the Nginx server. Figure 3.6 shows a diagram of our system architecture. When the Nginx server receives an HTTP request, it makes a simple decision:

- if the request is for a URL beginning with `/static/`, Nginx serves the static file itself from the `static/` directory in the project codebase;
- otherwise, it passes the request to the golang web server running on port 8080 and returns its response.

There are numerous advantages to this approach. For example, Nginx handles SSL

[4]`https://www.linode.com/`

termination, caching and gzip compression out of the box. It also keeps access and error logs and offers handy features such as URL rewriting. By allowing Nginx to do all of the heavy lifting, we can keep our golang web server very simple: it is responsible only for serving dynamic requests that depend on the user's session. This means that all of the code behind our golang web server is specific to our application, and no time has been spent reinventing the wheel to do the work that we can ask Nginx can do for us. This approach also has security benefits: our golang 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. Nginx is responsible for binding the privileged ports – its master process is run as `root`, but the worker processes (those that handle HTTP requests) run as the non-privileged user `www-data`.

Scalability Considerations

We mentioned on page 45 that Go's ease of deployment may become useful in the future in terms of scalability. This is because a Go program compiles to a single statically-linked binary. There is no dependency chain and no need to worry about shared libraries. At present, our dual-core Linux machine is adequate for the amount of load that the BrowserAudit application puts on the server. We have tested this by running BrowserAudit in multiple different browsers on multiple computers simultaneously; the runtimes were always acceptable and system load never reached unacceptable levels. If the application were to become popular then a single dual-core machine would not be adequate and we would need to expand our architecture. We have considered this already so that we do not face any scaling problems in the future.

The simplest way in which our application can scale is by upgrading the server to one with more computational resources. Due to their concurrent natures, both Nginx and our golang server (using the `GOMAXPROCS` environment variable) can make use of as many CPU cores as are available. This means that performance and request throughput can be improved simply by upgrading the server on which the application runs. This is not true scalability, however, and has an upper bound – we must instead consider how we can scale our application by running it across multiple servers. Nginx can very easily be configured as a load balancer, using multiple upstreams to serve requests. These upstreams could be multiple servers running our golang web server. Due to Go's ease of deployment, it would be very easy to deploy golang web server instances as necessary. In order to maintain session persistence, we would use Nginx's `ip_hash` directive to ensure that all requests from the same IP reached the same golang server. We could alternatively modify the golang processes to use a centralised session store.

3.2.3 Supervisor

We use the Supervisor daemon `supervisord`[5] to autostart and manage the golang server process. This is much simpler than writing `rc.d` scripts manually and is also easier to maintain. Supervisor also ensures that the process will be restarted in the event of a crash – something that cannot be achieved with `rc.d` scripts and would otherwise need to be written into the Go program itself.

Supervisor automatically starts the golang web server on boot, restarts it on failure, and redirects the server's `stderr` and `stdout` to log files. These logs are automatically rotated to preserve disk space. `supervisord` comes with its own `init.d` script to ensure that it starts when our Ubuntu server boots (and therefore that it starts our golang process).

3.3 Testing

It is not easy to test the implementations of our tests. A test with a small bug in it could still show as a pass in the browser, which is likely the result we are expecting and so the problem may never be detected. There are no easily-accessible browsers that intentionally contain security flaws so that we can test the negative cases. We considered modifying some of the open-source browsers to disable or break their security features in order to ensure that our tests fail when expected, but this proved to be too much work for the timescale of this project and is an idea that could perhaps be further explored in the future.

In many cases we instead test the negative cases of tests whilst implementing them, by temporarily changing something on the server-side. One example where this was possible is the *HttpOnly* cookie test that checks that an *HttpOnly* cookie set by the server is not accessible by JavaScript (see Section 5.4.1). The server normally sets a cookie with the *HttpOnly* flag set, but we can temporarily change this to be a cookie *without* the flag set and check that the test then fails in all browsers (JavaScript will be able to access the cookie when the flag is not set, but the test expects that it will not be able to). We are able to test many of our tests using methods similar to this, but we acknowledge that this is not a sustainable testing method.

Where possible, we test the negative cases by finding browsers that we know should fail our tests. Using sites like *Browserscope* and *CanIUse* (both of which were discussed in Section 2.3), we can obtain data about which browsers implement a certain security feature. We can choose browsers that don't implement a feature and then ensure that the BrowserAudit tests for that feature fail in those browsers. Likewise, we can do the

[5] `http://supervisord.org/`

opposite of this to test the positive cases, i.e. the tests that we expect to pass. We can select all browsers that reportedly implement a given security feature and then ensure that the relevant BrowserAudit tests pass in those browsers. Generally speaking, this allows us to test our BrowserAudit tests with a reasonable level of accuracy, although it is important to pay attention to any known quirks in the browser implementations of each feature that we test. We must also not rule out the possibility of a browser's implementation of a feature being buggy!

3.3.1 Selenium WebDriver

Selenium WebDriver[6] is a tool designed to automatically verify that web applications behave as expected. As a proof of concept, an example in the documentation automatically uses Mozilla Firefox to run a Google search. It waits 10 seconds for Google's JavaScript dynamic search result generation to complete and then returns the page's title. Selenium WebDriver tests are written in Java, so assertions (or even JUnit) can be used to state what the expected page title is and throw errors if the actual title does not match.

We should be able to use this in the future to automatically test BrowserAudit on browsers whose expected security test results we already know. Automatically running Selenium tests after a major change to our project would provide a sanity test that the functionality of our application is still correct. WebDriver hooks into many major web browsers: Mozilla Firefox, Internet Explorer, Google Chrome, Opera, iOS, and Android.

[6] `http://docs.seleniumhq.org/projects/webdriver/`

4 Implementation

In this chapter we discuss general implementation topics such as our web server setup and Go packages we use. We explain improvements we have made to Mocha – the client-side testing suite – to make it suitable for our needs. We also describe some common code that applies to all of BrowserAudit's security tests. The discussion of the implementations of the tests themselves, and how we have been able to automatically test browser security features, is saved for Chapter 5.

4.1 Nginx server

Our project makes use of four key domains: `browseraudit.com`, `test.browseraudit.com`, `browseraudit.org` and `test.browseraudit.org`. These are necessary in order to ensure a good coverage of various security features that involve cross-origin testing. We have a single SSL certificate that is valid for the four domains[1]. Nginx is 0 for handling all of these domains. As such, we define multiple `servers` in our Nginx configuration file. The overall server configuration is as minimal as possible – each URL is only available on the domains and schemes that are required.

Listing 5 shows an example `server` section from our Nginx configuration file. We have multiple of these so that we can serve a different subset of URLs for each domain, keeping the setup as minimal as possible. `listen` specifies the port to listen on (recall that 80 and 443 are the web ports for `http:` and `https:` respectively). The `listen` statement on line 3 tells the server to also listen on our server's IPv6 address. `server_name` specifies the domain name for the server. SSL is turned on since this particular server is running on port 443. All other SSL configuration options (such as the paths to the private key and certificate) are placed in a more global `http` context so that we don't have to repeat them in each server. Since we have a single SSL certificate covering all of our domains, all of these configuration options are the same for every server running on port 443.

The `location` directives specify the routing system described in Section 3.2.2 – static files are served directly by Nginx, whilst dynamic requests are passed to the golang server

[1]The certificate is also valid for `www.browseraudit.com`, which currently redirects to the BrowserAudit homepage. `www.browseraudit.org` also exists and redirects to the project homepage, although this is not covered by our SSL certificate

running on port 8080 using the `proxy_pass` directive. Note that Nginx uses the longest matching route, so the directive for `/static/` supersedes the one for `/`. We add a custom `X-Scheme` header to requests for `/set_protocol` so that the golang web server knows the scheme (`http:` or `https:`) used in the original request; Nginx does not pass this by default, but it is needed in the golang server for the HTTP Strict Transport Security tests (see Section 5.6.2). We intercept any proxy errors with the `proxy_intercept_errors` so that, in the case of a 404 Not Found error in a request to the Go server, we can catch this and instead redirect to the project homepage. We do this in case a user mistypes a URL, since the default golang HTTP server's 404 error is not especially user-friendly.

```
server {
  listen 443;
  listen [::]:443;

  server_name browseraudit.com;

  ssl on;

  location / {
    proxy_pass http://127.0.0.1:8080;
    proxy_intercept_errors on;
    error_page 404 = @homepage;
  }

  location = /robots.txt {
    alias /path/to/browser-audit/robots.txt;
  }

  location = /set_protocol {
    proxy_pass http://127.0.0.1:8080;
    proxy_set_header X-Scheme $scheme;
    proxy_intercept_errors on;
    error_page 404 = @homepage;
  }

  location /static/ {
    alias /path/to/browser-audit/static/;
  }

  location @homepage {
    rewrite  .*  https://browseraudit.com/ permanent;
  }
}
```

Listing 5: An example `server` section from our Nginx configuration

4.2 Go server

As described in Section 3.2, our application uses a custom web server written in Google's golang. This web server is responsible for handling any dynamic requests – those that depend on the user's session and are not just static files such as JavaScript and images.

4.2.1 Gorilla Web Toolkit

We use the popular Gorilla web toolkit[2] on top of golang's `net/http` package. It is important to note that this is not a framework but a toolkit – Gorilla is lightweight and implements the standard interfaces such that it can be used in conjunction with the built-in Go HTTP server in `net/http`.

We use two packages from the Gorilla toolkit: `gorilla/mux` for URL routing, and `gorilla/sessions` for session management.

URL Router

We use Gorilla's `gorilla/mux` package, which is a powerful URL router and dispatcher. This makes it simple for us to determine to which handler function a request should be passed. The `mux` package also allows us to extract variables from paths, which can then be accessed from within the handler functions.

```go
r := mux.NewRouter()

r.HandleFunc("/csp/{id:[0-9]+}", CSPHandler)
r.HandleFunc("/csp/pass/{id:[0-9]+}", CSPPassHandler)
r.HandleFunc("/csp/fail/{id:[0-9]+}", CSPFailHandler)
r.HandleFunc("/csp/result/{id:[0-9]+}", CSPResultHandler)

http.Handle("/", r)
```

Listing 6: Using `gorilla/mux` to route and dispatch requests

Listing 6 shows an example of how we use `gorilla/mux` from the Gorilla web toolkit to easily dispatch URLs for our Content Security Policy tests. We can see that it is easy to match URLs and pass them to the relevant handler functions, which compute and serve the responses. We also make use of path variables – the CSP test ID is extracted from the URLs where it can be easily accessed inside the handler function. Regular expressions are used to specify that the ID must consist of one or more digits. If no

[2] `http://www.gorillatoolkit.org/`

matching route is found, a 404 error is returned which will be intercepted by Nginx. The call to `http.Handle()` on line 9 is a call to Go's native HTTP library – it has nothing to do with Gorilla.

Sessions

Our application uses sessions to keep track of some test results and other test-related data for each user whilst their tests are in progress. We originally hoped that this would not be necessary, but it soon became apparent that sessions were indeed needed. One reason for this is that, 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 script running in the browser. In many of these cases, the client-side of our application running in the browser must send an additional request asking the server what the rest result was, so that it can be displayed to the user. In order for this to work, the server must remember some of each user's test results for the duration of their visit to our website. Where possible, this additional "what was the test result?" request is avoided and the result is sent back to the client-side with an earlier request. However this is not always possible due to the same-origin policy, hence the requirement for sessions. In other cases, it is still the client-side that decides on the test result, but sessions are required so that the client-side can find out information from the server about an earlier request it made, such as which cookies it sent.

We use Gorilla's sessions package to store our session data. More specifically, we use a `FilesystemStore` which stores the session data on the server filesystem. We store the session data in our filesystem rather than inside the session cookie itself due to concerns with the maximum cookie size supported by some browsers. Our session stores the results of hundreds of our browser tests, and so we want to avoid any potential problems with overly large cookies. Note that the `gorilla/sessions` package encrypts cookies, so the cookie value becomes much larger than the data stored in it.

4.2.2 Caching and the DontCache Function

In many of our Go handler functions (the functions that handle an HTTP request and generate a response) there is a call to a function `DontCache()`, usually at the top of the handler. This is a function that we have written ourselves, displayed in Listing 7. Its role is to ensure that an HTTP response is not cached, either by the receiving browser or any caching mechanisms in between the server and browser. The function takes one argument, a pointer to the `ResponseWriter` for the response, and sets the three headers necessary to disable caching.

```go
func DontCache(w *http.ResponseWriter) {
  (*w).Header().Set("Cache-Control", "no-cache, no-store, must-revalidate") // HTTP 1.1
  (*w).Header().Set("Pragma", "no-cache")                                   // HTTP 1.0
  (*w).Header().Set("Expires", "0")                                         // Proxies
}
```

Listing 7: Our `DontCache()` Go function that ensures a response is not cached

The use of the `DontCache()` function is very important due to a pattern commonly-used in our browser 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 our tests multiple times 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 calling `DontCache()` in the necessary handler functions.

4.3 Mocha

Recall that Mocha is the in-browser JavaScript testing framework that we use to simplify our implementation. It provides us with a means of automatically running “unit tests” and reporting the results to the user. In this section we discuss code we have written related to Mocha that is not specific to any particular browser test. We also discuss some design patterns used when writing our Mocha tests.

4.3.1 Tests

Our tests are executed by the page at `https://browseraudit.com/test`. This is served from `test.html` in the project codebase. It is important to note that our test page is loaded over HTTPS, not plain HTTP. We do this because it makes the implementations of the majority our security tests simpler. There are also the obvious security and privacy benefits from loading as much of our application as possible over secure transport. That said, loading the test page over HTTPS also partially limits our testing scope. This is discussed as a limitation in Section 6.2.2. Our Mocha JavaScript tests are written in `/static/js/test/*.js` files. Having a test directory is recommended as a best practice in the Mocha user guide. Since these are static files, we store them in our `static/` directory, meaning that Nginx will serve them directly without the requests ever reaching our golang web server.

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. Tests are written with the `it()` function.

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

Listing 8: Writing a test inside categories with Mocha

Listing 8 shows an example of how Mocha tests might be categorised in our case. We have a section for HTTP Strict Transport Security (HSTS), which sits underneath an HTTP Response Headers category. Inside the HSTS category, we have a single test. At the bottom of this test, `done()` is called, indicating that the asynchronous test is complete. For a synchronous test, the `done` argument can be removed from the anonymous function passed to `it()`. This means that it then does not need to be called upon completion of the test.

The test in Listing 8 does not do anything – it will always pass. This is because we have not written any assertions. Since we are using Chai with the Expect interface, we write our assertions with the `expect()` function. A test will fail if any of its assertions fail. In fact, a test will cease to execute as soon as an assertion fails. For this reason, our tests only contain a single call to `expect()`. In a case where a test might otherwise test multiple things (with multiple `expect()` calls), we instead split this into multiple individual tests, each with only a single assertion. This increases our test coverage, making sure that as much as possible is tested.

Callbacks vs. Timeouts

When writing the Mocha JavaScript tests, we try to use callbacks over timeouts wherever possible. This is because we want to avoid a situation in which we have to estimate the maximum amount of time it will take for the browser to load, for example, an `<object>` tag on a page. We want to avoid this because it would result in an implementation using a timeout, which must be for a predefined amount of time. Situations could arise in which the timeout is exceeded (for example, due to a user's slow connection or high load on our server), in which case a test may erroneously report an incorrect result.

Callbacks are the natural way to program in JavaScript. In many cases, we use jQuery's `.on("load", function() {...})` to register a callback to execute code after an element has been loaded[3]. This works for any element associated with a URL, notably images and frames, which are loaded dynamically many times in our codebase. In most cases we can use callbacks to avoid any need for manually-defined timeouts, ensuring that our tests always work as expected.

There are, however, cases in which we have been unable to find a callback solution. Many of these cases occur in the Content Security Policy tests, discussed in more detail with the implementation in Section 5.2 and in our evaluation in Section 6.2.1. One example case can be described as follows:

- the test page loads a frame from `frame.html`;
- `frame.html` loads an `<object>` element from `data.swf`;
- the test page wants to execute JavaScript code *after* the object has been loaded;
- however, a load callback on the frame triggers *before* the loading of the object.

Our solution for this at the moment is to set a timeout on the test page to execute 300ms after the loading of the frame. This is far from ideal; the timeout value will almost certainly need to be increased in the future in order to cater for slow collections. This has the major disadvantage that many browsers will be forced to wait even if the object was quickly loaded. Despite all of this, there is still the risk that the object hasn't been loaded at all even after a lengthier timeout! This is why callbacks are preferred wherever possible.

Callback behaviour in cases like the example above are not consistent across all browsers. Some can be fixed by modifying the `Content-Type` header served with the object being loaded from the page inside the frame. Unless we are certain that the callback solution works, we currently use a timeout. This is something that we hope to avoid in the future.

4.3.2 BrowserAudit Reporter

We have written our own Mocha "reporter" to display the BrowserAudit security test results in the browser. Reporters are Mocha's mechanism for modifying the style of its output without needing to modify any of the source code related to Mocha's core functionality. There are many terminal-based Mocha reporters for use with server-side

[3]jQuery's simpler `.load()` syntax for the same purpose has been deprecated, to avoid ambiguity to do with its method signature. The deprecated syntax is still used in many places in this report to improve the readability of code listings

(node.js) JavaScript, although we found very few in-browser Mocha reporters. It was necessary to write our own reporter in order to style the Mocha output to match our design that we discussed in Section 3.1.2. Our design is very different to the default Mocha in-browser design, so it was not feasible to simply use the default reporter with modified CSS classes. Our reporter is defined in `static/js/BrowserAudit.js`. We tell Mocha to use this reporter rather than the default by setting `reporter: BrowserAudit` in the `options` object that we pass.

Mocha reporters are constructed with an argument `runner`. This can be used to register event callbacks, which we make heavy use of in our reporter. For example, one can define a callback for whenever a test ends using `runner.on("test end", function(test) {...})`. The `test` argument here is a full JavaScript object representing a test, with properties such as `test.title`, `test.state` (e.g. "passed") and `test.err` which provides lots of important information regarding a test failure. Below is a complete list of the Mocha callbacks that we make use of in our reporter:

- "suite" – called at the start of a test suite, i.e. at the start of a category defined with `describe()`;
- "suite end" – called at the end of a test suite/category;
- "fail" – called on a test failure;
- "test end" – called when any test completes.

Thanks to these callbacks, the reporter implementation consists primarily of code to manipulate the DOM. We must create many DOM elements in order to display the test results. Below is a brief description of the DOM manipulation that occurs during each of the above callbacks:

- "suite" – if this is a root-level category, create a coloured panel representing the category of tests. If the category has a textual description, append this to the top of the panel. If this is a second-level category, add a heading (and description, if it exists) underneath its parent panel and a table for the test results to be stored in;
- "suite end" – if this is a root-level category, remove the loading icon that was displayed in the panel heading;
- "fail" – no DOM manipulation, this is all handled by the "test end" handler;
- "test end" – append the test result to the relevant results table. Update the result counts and progress bar at the top of the page. If a warning or failure, update the panel heading warn/fail counts and background colour if necessary. If this is

the last test, update the background colour of the results box at the top to be the green, yellow or red depending on the worst test result encountered.

Mocha also has an "end" callback designed to be called once all tests (and suites) are finished. We discovered that this does not work as it should in Internet Explorer, so we instead detect the "end" event by counting the number of tests completed and comparing it to the total number of tests (`runner.total`) on each "test end" event.

We use jQuery to simplify the DOM manipulation. In order to keep track of the category hierarchy (i.e. parent and child categories of tests) we use a stack, represented as an array in JavaScript. We push categories onto the stack on the "suite" event and pop them off on a "suite end" event. We also make use of the peek operation when adding a test result or second-level test category to the DOM. In order to find the DOM element to which these new elements should be appended, we locate it on top of the stack. Due to bugs in Internet Explorer and Safari when storing references to DOM elements on a stack, the stack actually stores string IDs of the elements they represent. These can be converted back to element references easily using `document.getElementById("myId")` (or jQuery's `$("#myId")` in our case, to keep the code style consistent with the rest of the file).

Alerts

Mocha only supports a test result of pass or fail, however our design incorporates two different test failure states: warning and critical, meaning that there are three possible test outcomes overall.

We could have forked the Mocha project and modified it to produce our own version that supports two different kinds of failure. We decided that this was unnecessary; maintaining a fork would mean that we would have to merge any future Mocha updates into our own repository to benefit from them. A better solution would be one that requires no change to the Mocha codebase. We achieved this by introducing a standard whereby any test whose title begins with the string `<warn>` should result in a warning on failure, as opposed to being flagged as a critical failure. This is implemented entirely in our custom reporter, and so will not be affected by any future Mocha updates. We simply check for the special string when a test ends, record whether or not a failure should be flagged as a warning or critical, and remove the string from the test title before it is displayed to the user.

Category Descriptions

Much like the alert failure status, Mocha has no support for displaying textual descriptions of a test suite beyond a title. Recall that our design allows for multiple paragraphs of text describing a block of tests. Following similar logic to that alerts, we did not want to have to modify the Mocha codebase to add support for this. We have instead implemented a standard in which a test category title can be of the form `name<|>desc` where `name` and `desc` are the category name and description respectively. The reporter detects this and displays the test category accordingly. The description is assumed to be HTML. For convenience, we have written a function `browserAuditCategory()` that acts as a wrapper around Mocha's `describe()` function. This function is displayed in Listing 9.

```javascript
function browserAuditCategory(name, desc, f) {
  if (desc === "")
    describe(name, f);
  else
    describe(name+"<|>"+desc, f);
}
```

Listing 9: Simplifying our category description system with a wrapper to `describe()`

The function takes the category's name and HTML description as separate arguments. The description can be left blank if desired. The benefit of this function is that it keeps the test files clean of the implementation details of our technique to separate the category name and descriptions in a single string.

4.3.3 afterEach Frame Removal

`afterEach()` is a Mocha feature that allows us to provide a JavaScript function to be run after every test, or after each test in a specific category of tests. We can make multiple calls to `afterEach()`. If called in the global scope, the callback function will be executed after every single Mocha test (regardless of whether it passes or fails). Alternatively, we can place it inside a call to `describe()` (the function used to start a new category of tests), which means that the callback will only be executed after each test in that given category. We use `afterEach()` in both ways – we make use of it in the global scope as described below, and in a category-specific scope in our HTTP Strict Transport Security tests (see Section 5.6.2).

We use `afterEach()` to run a function after each and every BrowserAudit test. This callback uses the DOM to remove any frames created by the test. The JavaScript code achieving this is shown in Listing 10. The majority of our tests create at least one `<iframe>` element, which are no longer needed as soon as the test result is reported. We

```
afterEach(function() {
  $("iframe").remove();
});
```

Listing 10: Using `afterEach()` to remove all `<iframe>` elements after each test completes

remove them so that the DOM doesn't grow unnecessarily large. Before we did this, we noticed that performance of our application decreased as time elapsed: the time taken to execute each test appeared to increase as a result of the excessive number of frames that had been appended to the `<body>` element. Removing the frames after each test resulted in a noticeable performance increase.

4.3.4 Stack Issues

When using Mocha with a relatively large test suite as in our case, there are problems relating to its internal stack in Internet Explorer (even in the latest version) and older versions of Safari. Mocha keeps a stack because it uses recursion when running categories of tests. There is an open issue on the project's repository for this stack problem, which has existed since July 2012[4]. Nobody seems to understand quite what the problem is; we spent some time of our own trying to diagnose and fix it, but were unsuccessful and soon decided that our time would be better spent elsewhere. A workaround is suggested in the issue's comments, which we make use of. It uses Mocha's `afterEach()` function to set a callback that uses `setTimeout()` to cleverly cut the deep stack trace after each test. This appears to solve the problem in both Internet Explorer and Safari.

[4]`https://github.com/visionmedia/mocha/issues/502`

5 Browser Tests

In this chapter we discuss how it is possible for us to test the browser security features currently assessed by BrowserAudit. This is interesting because we must first determine how to test each feature, and then how we can do so automatically. We describe any pitfalls encountered along the way (often to do with quirks in certain browsers), and also explain the JavaScript implementations of the tests.

We test all of the browser security features covered as part of our background in Section 2.2. We carefully selected these for multiple reasons: we believe we have a good selection of both must-have security features and modern security features whose implementations are young and not necessarily widespread across all modern browsers. The tests for the must-have features (e.g. the same-origin policy) are both interesting and important because the features should be implemented in any browser. The tests for modern security features (e.g. HTTP Strict Transport Security) are interesting because browser implementations of these are more likely to contain bugs, since they have not existed for anywhere near as long. There will also be browsers that don't implement these features, which will be highlighted by a high number of failed BrowserAudit tests.

5.1 Same-Origin Policy

The technical background for these tests is covered in Section 2.2.1. Our same-origin policy (SOP) tests currently cover the SOP for DOM access, the *XMLHttpRequest* API, and cookies.

5.1.1 DOM Access

In all document object model (DOM) tests, we are testing whether one page can access the DOM of another. In each test we create parent and child frames, where the child is an `<iframe>` inside the parent. The parent frame is a hidden `<iframe>` element appended to the main testing page. In each test, one frame (either the parent or the child) tries to access the `document.location.protocol` property of the other. This is one of many DOM properties to which access should be restricted by the same-origin policy [17].

Recall that, except in the case of valid `document.domain` values, DOM access should be denied by the browser whenever the origins of the two pages are different. Since Internet Explorer doesn't compare ports when checking origins, we should only test origins that differ by scheme or host. There is a further complication, however: due to mixed content rules, a page loaded over HTTPS cannot contain a frame loaded over HTTP. This will be blocked by the browser, and so this unfortunately further limits the number of cases we can test. Since our main test page is loaded over HTTPS, we are unable to test any cases involving a frame loaded over HTTP. We can now only test the same-origin policy for DOM access where differing hosts are used to cause an origin mismatch. The four hosts we use are `browseraudit.com`, `test.browseraudit.com`, `browseraudit.org` and `test.browseraudit.org`. We test all possible permutations of these without setting `document.domain` values, ensuring that the SOP always blocks the access whenever the hosts are different.

As well as testing that cross-origin DOM access is ordinarily blocked by the browser, we also test that the `document.domain` property can be used to loosen the same-origin policy restrictions. We test a wide range of both legal and illegal `document.domain` values, testing that DOM access is always correctly allowed or blocked as expected. We do this just for the `browseraudit.com` and `test.browseraudit.com` hosts, using the following four `document.domain` values:

- *(not set)*
- `browseraudit.com`
- `test.browseraudit.com`
- `audit.com`

The tests are categorised into four sections:

- parent frame origin `https://browseraudit.com`
  child frame origin `https://test.browseraudit.com`
  child accessing parent
- parent frame origin `https://browseraudit.com`
  child frame origin `https://test.browseraudit.com`
  parent accessing child
- parent frame origin `https://test.browseraudit.com`
  child frame origin `https://browseraudit.com`
  child accessing parent

- parent frame origin `https://test.browseraudit.com`
  child frame origin `https://browseraudit.com`
  parent accessing child

In each of these four sections, we test a variety of (but not all) combinations of the `document.domain` values. We only test the combinations of `document.domain` parameters that could lead to interesting results. If we were to test all possible permutations of the four values used, many combinations would end up testing repeat behaviour that had already been tested by a previous combination. This results in good coverage of the same-origin policy for DOM access, once we accept that changing the host is the only way in which we can mismatch origins.

Templates

Due to the large number of tests, each of which uses a similar pair of parent/child frames, we make use of templating to dynamically generate the HTML for each frame. This saves us from having to produce two new HTML files for each additional SOP for DOM test that follows the pattern of a parent and child frame, where one tries to access the other. We make use of golang's `html/template` package for this. There are four templates in total, located in the `sop/` directory:

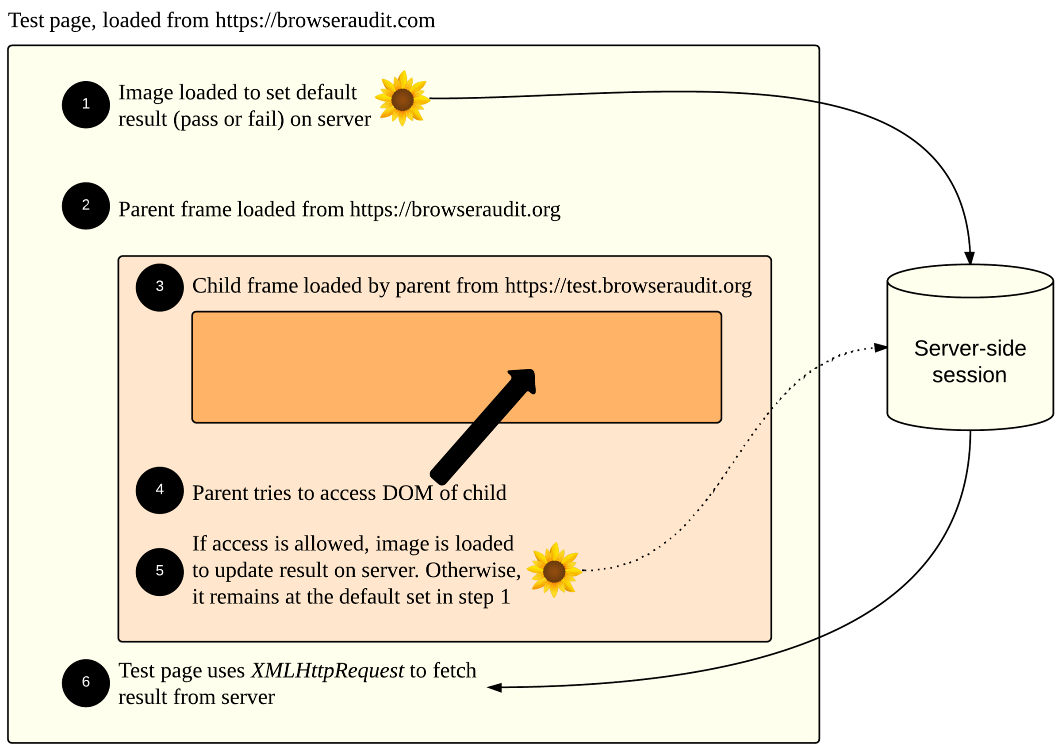
1. parent frame, parent accessing child (`p2c_parent.html`)
2. child frame, parent accessing child (`p2c_child.html`)
3. parent frame, child accessing parent (`c2p_parent.html`)
4. child frame, child accessing parent (`c2p_child.html`)

Templates 1 and 4 are the more interesting, since these are the templates in which the frame tries to access the DOM of the other. The variables found in the templates are described below:

- `{{.Script}}` – HTML, contains a `<script>` tag setting the `document.domain` property in many cases. For frames that do not set the property at all, this value is empty;
- `{{.Result}}` – string ("pass" or "fail"), what the result of the test should be if the DOM access is allowed;
- `{{.TestId}}` – integer, a unique ID for the same-origin policy test;

- `{{.ChildSrc}}` – URL, used in parent frames to set the source of their child frame.

Parent Accessing Child



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 5.1: An example of a SOP for DOM test in which the parent frame tries to access the DOM of its child

Figure 5.1 shows a high-level diagram of an example test in which a parent frame tries to access the DOM of its child. The parent is loaded from `https://browseraudit.org` whereas the child is loaded from `https://test.browseraudit.org`. We expect the access to be blocked since we are not setting any `document.domain` values in this test and the hosts are not the same. The first important thing to note is that the test result is saved on the server and queried later on. This is a pattern used often in our project to avoid the restrictions of the same-origin policy when automatically testing many features (including the SOP itself, as in this case). Note also that we are using images to set the results on the server. This is because there are no same-origin restrictions when loading images, so we can make a request to the `https://browseraudit.com/sop/pass/TEST_ID` and `/sop/fail/TEST_ID` URLs no matter what the origin of the page making the request. This is important because our session cookie is set for `*.browseraudit.com` so these requests must be sent to a page on either `test.browseraudit.com` or `browseraudit.com`. A page on one of the `.org` domains must still be able to update the result stored in the session, and so we use images to achieve this.

Listing 11 shows the template used to generate the parent frame in a DOM access test in which the parent frame tries to access the DOM of its child. This is template 1 in the list on page 63, and is comparable to the code behind the parent frame loaded in step 2 in Figure 5.1.

On line 5 of Listing 11 is the script (if any) to set the `document.domain` property of the frame.

On lines 13–15 is the code used to access the child's `document` object. This is more complicated than it perhaps should be due to differences in browser implementations. We first try to use the `contentWindow` property. If this is undefined, we use `contentDocument` instead. At this point, in some browsers `d` will be the child's `document` property whereas in others it will be the child's `window`. We must therefore check whether the property `d.document` exists, and set `d` to this if it does. We then know that our variable `d` references the child's `document` property with maximum possible browser support.

The parent then checks on line 17 whether it can access the property `d.location.protocol`. This is the child's `document.location.protocol` DOM property to which cross-origin DOM access should be blocked (unless the origins are the same or `document.domain` properties are being used to allow it). If access is allowed, we notify the server by requesting an image from the URL `/sop/{{.Result}}/{{.TestId}}`. When the server receives this request, it updates the result for that particular test in the session. If DOM access is not allowed, we do not need to do anything since a default result has already been set by the JavaScript on the main test page running all of the tests (step 1 in Figure 5.1).

We create the child frame with JavaScript on line 11 to ensure that its `onload` handler is always called. We can guarantee that this is the case by setting the child's `onload` property before its `src` attribute. Without doing this, it is possible that the child frame would be completely loaded by the browser before the `onload` event was even registered, and so it would not be executed. This is especially relevant to us since the contents of the child frame is very small and quick to render (a barebones HTML document, perhaps with some script to set the `document.domain` property).

Child Accessing Parent

The tests in which the child frame tries to access its parent frame are much simpler. There is no need to worry about dynamically creating frames or `onload` events. The

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  {{.Script}}
</head>
<body>
  <script>
    // We create the iframe with JavaScript to guarantee that the onload is always
    // executed
    var child = document.createElement("iframe");
    child.onload = function() {
      var d = (child.contentWindow || child.contentDocument);
      if (d.document)
        d = d.document;

      if (d.location.protocol) {
        var img = document.createElement("img");
        img.setAttribute("src", "/sop/{{.Result}}/{{.TestId}}");
        document.body.appendChild(img);
      }
    };
    child.setAttribute("src", {{.ChildSrc}});
    document.body.appendChild(child);
  </script>
</body>
</html>
```

Listing 11: Parent frame template in a SOP for DOM test in which the parent tries to access the child's DOM

template for the parent frame simply sets its `document.domain` property (if any) and has an `<iframe>` tag loading the child frame from `{{.ChildSrc}}`. In the child frame, which also sets any `document.domain` value in the usual way, the DOM of the parent can be accessed consistently across browsers using the `window.parent.document` property. If access is allowed, the child frame notifies the server in the same way as the previous example: by dynamically loading an image from a special URL.

JavaScript Test Functions

So far we have discussed the templates used to dynamically generate the frames used in these tests, but not how the variables required for the templates are passed to the server. This is done by our JavaScript testing code. We have two functions that run the SOP for DOM tests: `parentToChildSopTest()` and `childToParentSopTest()`. Each of these functions takes the following arguments:

- `desc` (string) – the textual test description passed to Mocha's `it()` function;

- `shouldBeBlocked` (Boolean) – `true` if and only if we expect the DOM access to be blocked;
- `parentPrefix` (string) – the prefix appended to the parent frame's URL, defining the origin from which it should be loaded, e.g. `https://test.browseraudit.com`;
- `parentDocumentDomain` (string) – the `document.domain` value (if any) of the parent frame;
- `childPrefix` (string) – the prefix appended to the child frame's URL, defining the origin from which it should be loaded;
- `childDocumentDomain` (string) – the `document.domain` value (if any) of the child frame.

An example of how one of these functions is called as part of our tests is shown in Listing 12. The parent frame has origin `https://test.browseraudit.com` whilst the child frame has origin `https://browseraudit.com`. Cross-origin DOM access could be allowed between these two frames if they each set their `document.domain` property to be `browseraudit.com`, however this is not the case. The parent frame sets its value to `browseraudit.com`. The child sets its value to `test.browseraudit.com` – not only is this an illegal value, but it also doesn't match the value set by the other frame. We therefore expect the DOM access to be blocked, so `shouldBeBlocked` is set to `true`.

```
parentToChildSopTest("...textual description...",
  true,                             // shouldBeBlocked
  "https://test.browseraudit.com", // parentPrefix
  "browseraudit.com",              // parentDocumentDomain
  "https://browseraudit.com",      // childPrefix
  "test.browseraudit.com");        // childDocumentDomain
```

Listing 12: A same-origin policy for DOM test using one of our two functions

It is hopefully clear that, through the use of our functions `parentToChildSopTest()` and `childToParentSopTest()`, it is very simple to write a new same-origin policy for DOM test following our pattern. All that needs to be provided are the origins and `document.domain` values (if any) of the two frames, what the expected result is, and a test title for the user. The code in these functions then carries out the following tasks:

1. a unique ID for the test is obtained. These start at 0 and are incremented with each new test;
2. a request is made to save the default result for the test (pass or fail) in the server's session[1]. This is the result that will be overwritten only if the DOM access is

[1]We originally had the Go handler for the parent frame (step 3) also setting the default result. This

allowed;

3. the parent frame is loaded. The source URL of this frame contains all information needed by the server to dynamically generate both the parent frame and its child;

4. once the parent frame is loaded (which, in turn, means the child frame has been loaded and the DOM access attempted), the test result is queried from the server.

When one of these tests fails, the message to the user only states that the expected result was a pass whereas the actual result was a fail. This could perhaps be improved by instead showing the user the `location.protocol` obtained by the frame initiating the DOM access. We would expect this to be undefined in tests where we expect the DOM access to be blocked, and set otherwise.

5.1.2 XMLHttpRequest

Recall that, in the same-origin policy for the *XMLHttpRequest* API, the `document.domain` property cannot be used to relax the policy's restrictions. This makes our test coverage for *XMLHttpRequest* simpler than it was for DOM access since we are only testing that cross-origin requests are correctly blocked. The other key difference when we compared the SOP for DOM and *XMLHttpRequest* was that Internet Explorer takes the port number into account when comparing origins for *XMLHttpRequest.* This is not the case for DOM access, when only the scheme and host are compared. This means that we could, in theory, test cross-origin requests with the *XMLHttpRequest* API where the origins differ port. At present, however, we only test origins that mismatch due to differing schemes and hosts.

We have 28 tests for the SOP for *XMLHttpRequest.* In each test, a frame is loaded from the source origin. Inside this frame is JavaScript that makes an *XMLHttpRequest* to a page at the destination origin. If this request reaches the server, the result for that test is updated in the session. Otherwise, the test result remains at the default value already set by our JavaScript test. We have 2 tests in which the *XMLHttpRequest* should be allowed (i.e. the origins are identical). The other 26 tests try a variety of origin mismatches (where the origins differ in scheme, host, or both) and expect the requests made with the *XMLHttpRequest* API to be blocked by the browser. We use all four hosts here: `browseraudit.com`, `test.browseraudit.com`, `browseraudit.org` and `test.browseraudit.org`. As in the DOM access tests, our testing scope has been slightly limited due to mixed content rules. We cannot test any requests from an `http:` scheme, since this would require us to be able to load an HTTP frame from an HTTPS frame. We

worked well for tests using `browseraudit.com` and `test.browseraudit.com`, but we had to split it into two requests after we introduced the `.org` domains. This is because our session cookie is tied to `*.browseraudit.com`, yet some parent frames are now loaded from `.org` domains can, however, attempt to make requests from an `https:` origin to an `http:` one (which, of course, should be blocked).

Once again, we make use of templating to generate frames dynamically. For the same-origin policy for *XMLHttpRequest* tests, we have a single template (`sop/ajax.html`) with just one variable `{{.Dest}}`. This represents the destination URL to which the *XMLHttpRequest* is made. A same-origin policy for *XMLHttpRequest* test can be executed with our `ajaxSopTest()` function, which takes four arguments: `desc`, `shouldBeBlocked`, `sourcePrefix` and `destPrefix`. The prefix arguments can be used to set the origins of the page making the request and the page being requested through the *XMLHttpRequest* API.

5.1.3 Cookies

When testing the same-origin policy for cookies, we test cookie scope using the *Domain* and *Path* attributes. As part of this, we test that cookies with illegal *Domain* attributes are correctly thrown away by the browser and not sent back to the server with any further requests.

Domain Scope

In each domain scope test, we first want the server to set a cookie (using the `Set-Cookie` header) with a given *Domain* value from a specific domain. The reason we care about the domain of the page setting the cookie is because some values will only be legal if served in responses from certain domains. For example, a page at `browseraudit.com` should not be able to set a cookie for domain `test.browseraudit.com`. We then want to make a second request, perhaps to a different domain, and detect whether or not the browser sent the cookie with this request. In some tests we expect the cookie to have been sent, whereas in others we expect that it isn't sent due to the same-origin policy. The basic pattern of a single cookie domain scope test is therefore as follows:

1. client-side JavaScript makes a request to a server-side page on some domain, asking it to respond with a `Set-Cookie` header with the requested name, value, *Domain* and *Path* (the path is always `/` for domain scope tests);

2. the JavaScript test makes a further request to the server, perhaps on a different domain. This is the request for which we want to know whether or not the browser sends the cookie set in step 1. The server stores the value of this cookie (if any) in its session;

3. in the JavaScript once more we make an AJAX request to server, requesting that it replies with the cookie (if any) sent in step 2;

4. for tests in which we expect cookie not to have been sent in step 2, we expect the response from the server to be “none”. Otherwise, we expect the value returned to be equal to the value we set in step 1.

The cookie names used in each test are unique on every run to ensure that multiple consecutive runs of our test suite can never interfere with one another. We ensure the uniqueness of the cookie names by concatenating the current timestamp with the test’s unique ID. The value of the cookie set in step 1 is also set to the current timestamp, and this is the value that we expect in step 4 in tests where we do not expect the same-origin policy to stop the cookie from being sent. The cookies have a short expiry time (1 minute) to stop the browser from quickly filling up with temporary cookies, especially if our test suite is run a few times in quick succession.

```
function domainScopeCookieTest(desc, shouldBeUnset, domainSetFrom, cookieDomain,
  domainAccessedFrom) {

  domainScopeCookieTest.id = domainScopeCookieTest.id || 0;

  var id = domainScopeCookieTest.id++;

  it(desc, function(done) {
    var timestamp = "" + new Date().getTime();
    var name = "sopscope"+id+timestamp;
    var value = timestamp;
    var domain = cookieDomain;
    var path = "/";
    $("<img />", { src: "https://"+domainSetFrom+"/sop/cookie/"+name+"/"+value+"/"+
      domain+"/"+$.base64.encode(path) })
    .load(function() {
      $("<img />", { src: "https://"+domainAccessedFrom+"/sop/save_cookie/"+name })
      .load(function() {
        $.get("/sop/get_cookie/"+name, function(c) {
          if (shouldBeUnset)
            expect(c).to.equal("none");
          else
            expect(c).to.equal(value);
          done();
        });
      });
    });
  });
}
```

Listing 13: Testing cookie scope with the *Domain* parameter

Listing 13 shows our JavaScript function `domainScopeCookieTest()` that implements the pattern described above. The request on line 14 is the request to the server to which the server will respond with a `Set-Cookie` header with the name, value, *Domain* and *Path* specified. In line 17, we make a request to the server asking that it saves the value of the cookie (if any) that is sent with the request. We then retrieve this value from the server's session in the request on line 20. Depending on the value of the `shouldBeUnset` Boolean argument, we either want this to be "none" or the value we set on line 11.

```
domainScopeCookieTest("...textual description...",
  false,                   // shouldBeUnset
  "browseraudit.com",       // domainSetFrom
  ".browseraudit.com",      // cookieDomain
  "test.browseraudit.com"); // domainAccessedFrom
```

Listing 14: An example call to `domainScopeCookieTest()`

We call the `domainScopeCookieTest()` function once for each of the domain scope tests. There are 19 tests using this function in total. An example of one of them is shown in Listing 14. In this particular test we expect that the cookie should be sent to the server, i.e. the same-origin policy for cookies should not apply. A page on `browseraudit.com` sets a cookie with domain `.browseraudit.com`, which means that it should be sent to `*.browseraudit.com` as well as `browseraudit.com` itself. We test this by checking that the cookie is sent with a request to `test.browseraudit.com`.

We make use of the `browseraudit.org` and `test.browseraudit.org` domains where possible. We have tests in which cookies are set by the `.org` domains and we then test that they are not sent to either `browseraudit.com` or `test.browseraudit.com`. We are unable to write any tests in which a cookie is *received* by one of our `.org` domains because our session cookie is scoped to `*.browseraudit.com`. A request to `browseraudit.org` requesting that the server remembers a cookie value would result in the cookie value on the server immediately becoming inaccessible – the server would respond from `browseraudit.org` with a brand new session cookie that it tries to set for domain `*.browseraudit.com`; this new session cookie is invalid (due to the domain mismatch) and will be thrown away by the browser.

Illegal Domain Values

Using the same `domainScopeCookieTest()` function, we also test the behaviour of illegal *Domain* attribute values. Examples include trying to set a cookie with *Domain* `test.browseraudit.com` from `browseraudit.com` and trying to set cookies with *Domain* `.com` (too broad). In all of these cases we expect that the cookies are immediately discarded by the browser and never sent back to the server.

Path Scope

We have two simple tests that test the scope enforced by a cookie's *Path* attribute:

- cookie with path `/sop/path/` should be sent to `/sop/path/save_cookie/*`;
- cookie with path `/sop/path/` should not be sent to `/sop/save_cookie/*`.

These tests use the exact same handlers on the server-side for setting a cookie, saving its value in the session, and retrieving this value from the session, as the domain scope tests we have seen already. Their JavaScript implementation is also very similar, except the *Domain* parameter is now fixed (`.browseraudit.com`) and the *Path* parameter is variable.

5.2 Content Security Policy

The technical background for these tests is covered in Section 2.2.2.

These tests aim to cover the Content Security Policy 1.0 specification [36]. Recall that, despite the CSP being standardised, a website must serve its pages with three different CSP headers in order to achieve maximum compatibility with older browsers. Note however that there is a reasonable possibility that a developer will use only the `Content-Security-Policy` header when implementing the CSP, since this is the header described in the standard. For this reason our CSP security tests use only the `Content-Security-Policy` header. A browser that implements the CSP using only an older experimental header will fail our tests – we will flag this as insecure since the browser does not meet the CSP standard.

Our tests offer good coverage of the CSP specification. Notable exceptions, for which we hope to write tests in the future, include:

- the `font-src` and `connect-src` directives;
- the `sandbox` directive;
- the `report-uri` directive and `Content-Security-Policy-Report-Only` header;
- origins that differ in scheme or port (we currently only check origins that differ in host);

- redirects – many of the CSP directives state that a browser can follow redirects when enforcing the policy. We do not currently have any tests involving redirects. It would be interesting to test what happens when an allowed source redirects to an unallowed source, for example.

Beyond these exceptions we have good coverage of the CSP: we have 126 Content Security Policy tests in total.

5.2.1 Lack of Templating

One of the more interesting aspects of the implementation of these tests, especially when compared with many of our other tests, is that we do not make any use of any server-side templating in our Content Security Policy tests. We opted not to use templates despite the fact that that all of these tests require HTML files, some of which are similar to each other.

Our reason for not using templates to generate the HTML files is that the files aren't quite different enough for templates to make a big improvement. For each group of HTML files that are similar, there are usually only a few such files, and each group of similar files is vastly different to every other group. This will hopefully become clearer when we discuss the actual implementations below. Whilst templates would reduce the number of HTML files in the codebase, as a proportion it would not be a significant reduction. We believe that templating the CSP tests in their current state would result in a much more complex implementation for little gain, and so it is not worthwhile at this point. This may not always be the case. For example, if we were to test origin mismatches not just in host but in scheme and port too, then there would be many more similar HTML files at which point templating may be a sensible option. Until we reach that point, having an individual HTML file for each test is what we found to be the best solution. We also only use the `browseraudit.com` and `test.browseraudit.com` domains in these tests. If we were to expand these to use the `browseraudit.org` and `test.browseraudit.org` domains, then templating would likely become a wise decision.

5.2.2 Detection of Features

There are some cases in which we test that the Content Security Policy correctly blocks the loading of a resource that a browser may not even attempt to load, either because the browser doesn't support a feature or because a plugin is missing. An example of this is testing which resources we can load with the `<object>` and `<embed>` tags (governed by the the `object-src` directive) when Adobe Flash Player is not present. This makes no sense – we can't test whether the browser's CSP implementation allows or blocks something if it will never even attempt to load it!

Our solution to this is to automatically detect whether or not a browser supports the relevant feature before running the CSP tests related to it. This means that we only test the CSP directives relevant to the features supported by a browser. In many cases we use the *Modernizr*[2] library to achieve this. This provides us with a global `Modernizr` object containing Boolean properties indicating support for many features. For example, before running the `media-src` tests that use the `<audio>` tag, we check for HTML5 audio support using `Modernizr.audio`. Another library that we use for feature detection is *SWFObject*[3], which is used to detect Adobe Flash Player.

5.2.3 cspTest Function

All Content Security Policy tests use the same function, `cspTest()`, whose source can be found in Listing 15. In each test we load a frame from a URL of the form `/csp/n`, where n is a test ID. This URL will be served by our golang server with a `Content-Security-Policy` header whose value is specified by the `policy` parameter passed in the URL. We also provide the Go server with a parameter `defaultResult`; this is the test result (pass or fail) that should be reported unless it is overwritten by a request made by the framed document. The contents of the frame is loaded by the golang server from `csp/n.html` and served with the specified `Content-Security-Policy` header.

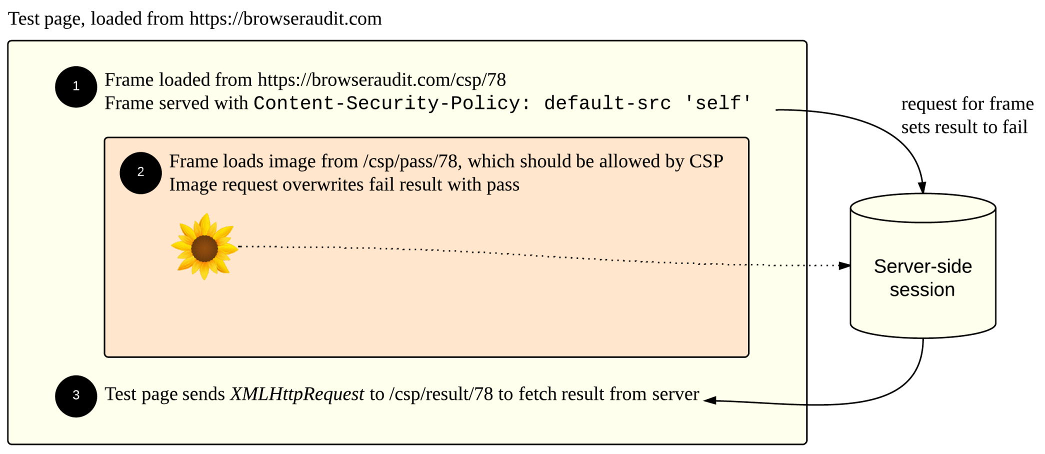

Test page, loaded from https://browseraudit.com
1
Frame loaded from https://browseraudit.com/csp/78
Frame served with Content-Security-Policy: default-src 'self'
request for frame sets result to fail
2
Frame loads image from /csp/pass/78, which should be allowed by CSP
Image request overwrites fail result with pass
Server-side session
3
Test page sends XMLHttpRequest to /csp/result/78 to fetch result from server

Figure 5.2: An example CSP test in which we are testing that an image is correctly allowed by the policy

[2] `http://modernizr.com/`
[3] `https://code.google.com/p/swfobject/`

```
1  function cspTest(desc, id, policy, shouldBeBlocked, opts) {
2    var policyBase64 = $.base64.encode(policy);
3
4    it(desc, function(done) {
5       var defaultResult = (shouldBeBlocked) ? "pass" : "fail";
6      if (typeof opts.timeout === "undefined")
7        $("<iframe>", { src: "/csp/"+id+"?policy="+policyBase64+"&defaultResult="+
8          defaultResult })
9        .css("visibility", "hidden").appendTo("body").load(function() {
10         $.get("/csp/result/"+id, function(result) {
11           expect(result).to.equal("pass");
12           done();
13         });
14       });
15     else
16       $("<iframe>", { src: "/csp/"+id+"?policy="+policyBase64+"&defaultResult="+
17         defaultResult })
18       .css("visibility", "hidden").appendTo("body").load(function() {
19         setTimeout(function() {
20           $.get("/csp/result/"+id, function(result) {
21             expect(result).to.equal("pass");
22             done();
23           });
24         }, opts.timeout);
25       });
26   });
27 }
```

Listing 15: `cspTest()` function used for all Content Security Policy tests

The basic pattern of every CSP test as defined by the `cspTest()` function is illustrated in Figure 5.2. Each of the files `csp/n.html` tries to load a resource in a way that can be governed by the Content Security Policy. We then test whether or not the browser sends a request to the server to load this resource. Depending on whether or not the resource loading should have been allowed by the server, we report the result of a pass or a fail to the user. In these tests there is no further information that we can provide which would be of interest to the user – the best we can tell him is whether his browser correctly allowed or blocked the resource as specified by the policy. We use the special URLs `/csp/pass/TEST_ID` and `/csp/fail/TEST_ID` to update the test result stored on the server-side for that specific test ID. These are the URLs that each `csp/n.html` file tries to load. The `cspTest()` function then requests the result on either line 10 or 20 of Listing 15 using the golang handler behind the `/csp/result/TEST_ID` URL.

Timeouts

The `opts` argument to `cspTest()` is an options object. At present, we only ever pass this object with one property: `timeout`. The `timeout` option allows us to specify a timeout in milliseconds that should be waited before the server is queried for the test result and the result reported to the user. This is an alternative to the default behaviour, which is to use a load event on the frame created to determine when to request the test result. As discussed in Section 4.3.1, we prefer the default behaviour (a load event callback) over a timeout wherever possible. This is achieved by passing an empty `opts` object `{}`. Unfortunately the timeout could not be avoided in some cases. We discuss these cases as limitations during our evaluation in Section 6.2.1.

5.2.4 Testing default-src

Our Content Security Policy tests are separated into sections based on the resource type (e.g. stylesheets and scripts) that we are trying to load. Each of these resource types is governed by its own CSP directive (`style-src` and `script-src` for stylesheets and scripts respectively). We must not forget the `default-src` directive, which specifies the default policy to be applied if no more specific directive is specified for a resource type. For each CSP test, we therefore test the policy both when specified with the specific directive for that resource type and when specified with `default-src`.

For example, when testing the Content Security Policy for a stylesheet loaded from `https://test.browseraudit.com`, we test two cases in which the browser should allow the loading of the stylesheet with the following `Content-Security-Policy` header values:

- `default-src 'none'; style-src https://test.browseraudit.com`
- `default-src https://test.browseraudit.com`

For a test like the above, we will also test that the stylesheet is correctly blocked when other directives are set. Using the same example, the negative cases are tested with the following header values:

- `default-src 'none'`
- `default-src 'self'`
- `default-src https://test.browseraudit.com; style-src 'none'`
- `default-src https://test.browseraudit.com; style-src 'self'`

The exact headers used depend on the origin of the resource being loaded. The key observation is that we test both `default-src` and the relevant specific directive (`style-`

`src` in this case). When testing cases that we expect to be blocked by the resource-specific directive, we ensure that the `default-src` directive would otherwise allow the resource to be loaded. This tests that the browser prefers specific directives over the default directive.

5.2.5 Stylesheets

We test the loading of stylesheets in two different ways: a `<link>` tag in the head of an HTML document, and using CSS's `@import` rule from within an inline style block.

<link> tag

These tests load stylesheets in the standard way used by most websites, with a `<link rel="stylesheet" href="..." />` HTML tag in the head of the document. The URL in the `href` attribute will be either `/csp/pass/TEST_ID` or `/csp/fail/TEST_ID` depending on whether we expect the CSP to allow or block the loading of the stylesheet.

@import

In these tests we use the `@import` CSS rule to load the stylesheet. This rule provides a way of loading a stylesheet from within another stylesheet. Listing 16 shows how this is done. Note that we place our `@import` rule inside an inline style block. This means that we must have `'unsafe-inline'` in our CSP directives to ensure that the `@import` rule is considered by the browser. We are not testing the behaviour of `'unsafe-inline'` here, only the CSP for stylesheets.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <style>
    @import url("/csp/pass/30");
  </style>
</head>
<body>
</body>
</html>
```

Listing 16: Loading a stylesheet with `@import` to test the CSP for stylesheets

Coverage Improvements

We could improve the coverage of our stylesheet tests by testing the `<link />` tag when placed in the document's body instead of the head. We could also load the stylesheets using JavaScript, creating the `<link />` tag dynamically and appending it to the head or body. One further improvement that could be made to the test coverage for the `style-src` directive is stylesheet chaining: using `@import` to load a stylesheet from within a stylesheet that has already been loaded either with a `<link />` tag or an `@import`.

5.2.6 Scripts

We test the Content Security Policy for scripts (governed by the `script-src` directive) in two ways: `<script>` tags and the `Worker` and `SharedWorker` APIs.

<script> tag

The `<script>` tag is the standard way of loading a JavaScript file. The tag can be placed in both a HTML document's head and body. At present, we only test script loading when the `<script>` tag is placed in the document's body.

Worker and SharedWorker

A more interesting way in which we test the CSP for scripts is by using the JavaScript APIs for `Worker` and `SharedWorker` construction. Web workers are JavaScript scripts that run in the background of a page. They are designed for computationally expensive tasks that can run in the background in parallel with the page's main script. The `Worker` interface spawns real operating system threads, taking advantage of multi-core systems.

Web workers are not supported across all browsers, and so we detect that the browser supports them before running the tests. We use *Modernizr*'s `Modernizr.webworkers` and `Modernizr.sharedworkers` Boolean properties for this. We originally only used the `Modernizr.webworkers` property but later realised that some versions of Safari and older versions of Firefox implement workers but not shared workers, so we now test for support as if they are two different features[4].

[4]The key difference between a `Worker` and a `SharedWorker` is that a `Worker` can only be accessed by the script that created it, whereas a `SharedWorker` can be accessed by any script from the same origin. Maybe we could write BrowserAudit tests for this in the future!

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <script>
    new Worker("/csp/pass/46");
  </script>
</head>
<body>
</body>
</html>
```

Listing 17: Loading a script with a `Worker` constructor to test the CSP for scripts

Workers are constructed with a path to the JavaScript file from which the script should be loaded. This is the means of loading a script that we test is protected by the CSP. Listing 17 shows how we construct a web worker when testing the CSP in this manner. Note that we are making use of an inline script block, so all web worker tests must use `'unsafe-inline'` in their CSP directives.

When implementing the `Worker` and `SharedWorker` tests we encountered some interesting behaviour in older verions of Safari. Upon trying to load the worker from a URL such as `/csp/pass/46`, the page would reload. By this we mean the entire test page, not just the frame that tries to load the worker. This would result in a redirect loop, eventually causing Safari to give up on loading the page. We discovered that this was due to the empty response body being served by the server. When we try to load a worker from a URL with no response body, it causes Safari versions 6 and older to reload the page. We fixed this by serving a JavaScript file containing `"//"` for the relevant pass and fail URLs, i.e. a JavaScript line comment. This stops the redirect issue from occurring.

Coverage Improvements

Where we load a script by constructing a web worker, we could place the JavaScript code that constructs the worker in a JavaScript file of its own, which would be loaded in a `<script>` tag in the normal way. In all cases where we load a script with a `<script>` tag, we could place this tag in the document's head instead of the body. We could also dynamically create the script tag in JavaScript itself, another method of using JavaScript to load a script file.

5.2.7 ‘unsafe-inline’

We have used the `'unsafe-inline'` keyword in CSP directives for tests seen so far, but only as a means of enabling inline `<script>` and `<style>` blocks to test the `script-src` and `style-src` directives. We have not yet tested the behaviour of `'unsafe-inline'` itself. BrowserAudit’s `'unsafe-inline'` tests cover the four key aspects of the directive’s behaviour: inline `<script>` tags, inline `<style>` tags, inline `onload="..."` event handlers, and inline `style="..."` attributes.

Inline <script> tags

See Listing 18 for an example of how we test whether or not an inline `<script>` block is executed. This is done with a simple redirect – if the inline script block is executed then the JavaScript on line 8 will redirect the frame, making a request to a URL that updates the test result on the server.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8" />
5  </head>
6  <body>
7    <script>
8      window.location = "/csp/pass/3";
9    </script>
10 </body>
11 </html>
```

Listing 18: Testing the CSP `'unsafe-inline'` keyword by detecting whether an inline `<script>` block is executed

Inline <style> tags

We test inline `<style>` tags in a similar way to inline `<script>` tags. The style block used by this technique is shown in Listing 19. We use CSS to set the background of the page body to be an image loaded from the server. If the server receives a request for this image then we know that the inline style block has been rendered by the browser.

```
<style>
  body {
    background: url("/csp/pass/7");
  }
</style>
```

Listing 19: Testing the CSP `'unsafe-inline'` keyword by detecting whether an inline `<style>` block is rendered

Inline onload event handlers

Our inline event handler tests use a JavaScript redirect just like we saw in Listing 18 for testing inline `<script>` blocks. We attach the onload event to the body of our testing page, so the code looks like `<body onload='window.location = "/csp/pass/5"'>`. This allows us to detect whether or not the browser has executed the inline `onload` event.

Inline style attributes

We test inline style attributes in a very similar manner. We attach a `style` attribute to the document body, which tries to load a background image using the same method as in Listing 19. An example of the resulting code is `<body style='background: url("/csp/pass/9")'>`.

5.2.8 'unsafe-eval'

The `'unsafe-eval'` keyword tells the browser to allow strings to be executed as code. The most common method of doing this in JavaScript is through the `eval()` function, however there are other methods. In total, we test four methods of evaluating strings as code: the `Function` constructor, `eval()`, `setTimeout()`, and `setInterval()`. In all of these `'unsafe-eval'` tests, we use a JavaScript redirect in order to detect whether or not the browser evaluated the string as code. If the `'unsafe-eval'` keyword is not present then we expect the execution of the code string to be blocked by the browser.

Function constructor

The `Function` constructor allows a new JavaScript function to be created, where the function body is given as a string of JavaScript code. For example, one can create a function `add()` with `var add = new Function("x", "y", "return x + y")` which can then be called as `add(3, 7)`. Note that `"return x + y"` is a string which is evaluated as code.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8" />
5    <script>
6      var f = new Function('window.location = "/csp/fail/12"');
7      f();
8    </script>
9  </head>
10 <body>
11 </body>
12 </html>
```

Listing 20: Testing the CSP `'unsafe-eval'` keyword with the `Function` constructor

Listing 20 shows how we use a redirect in a function created with `Function` in order to test `'unsafe-eval'`. As with many other CSP tests seen so far, we must also use `'unsafe-inline'` to allow the inline script block on lines 5–8.

eval()

We test `'unsafe-eval'` with the most traditional method, `eval()`, using a simple call `eval('window.location = "/csp/pass/13"')`. This is placed in an inline script block and allowed with `'unsafe-inline'` just as in Listing 20.

setTimeout() and setInterval()

The `setTimeout()` and `setInterval()` JavaScript functions are both methods of scheduling code to be executed at some point in the future. `setTimeout()` schedules code to be executed exactly once after some timeout n milliseconds, whereas `setInterval()` schedules the code to be executed every n milliseconds. The code can be passed as a function or as a string of code to be evaluated and executed. The latter of these methods involves evaluation and so is the one in which we are interested.

See Listing 21 for two examples of how we try to evaluate code strings whilst testing the `'unsafe-eval'` CSP keyword. As always, we must also set `'unsafe-inline'` in our `Content-Security-Policy` headers for these tests to allow the inline script blocks to be executed. These tests require a timeout – we cannot use a load event on the frames containing the script blocks in Listing 21. The `setTimeout()` and `setInterval()` functions schedule code to be executed *after* the frame loads. We must therefore use a

```
<script>
  setTimeout('window.location = "/csp/pass/15"', 100);
</script>
```

```
<script>
  setInterval('window.location = "/csp/fail/18"', 100);
</script>
```

Listing 21: Testing the CSP `'unsafe-eval'` keyword with the `setTimeout()` and `setInterval()` functions

timeout in the testing code to check the test result at some point after the 100ms delay caused by `setTimeout()` and `setInterval()`. We currently use a 300ms timeout in our call to `cspTest()` for this.

5.2.9 Objects

We test the CSP directive `object-src` for objects with the `<object>` and `<embed>` HTML tags. We encountered an interesting issue here with Chromium-based browsers such as Google Chrome. Listing 22 shows an example HTML document used in an `object-src` test. Note that as well as trying to load an object from the URL `/csp/fail/37` with the `data` attribute, we also set the `type` attribute. We explain this with a comment in the code on lines 7–9 because it is a strange case. Without the `type` attribute, Chromium-based browsers treated the loading of the object as the loading of a *frame*. Consequently, the browser looks at the `frame-src` CSP directive as opposed to `object-src`. We test `frame-src` in a separate set of tests (see Section 5.2.12); we are testing `object-src` here, not frame `frame-src`. We found a way to force Chromium-based browsers to treat our URL as an object by explicitly setting the `type` attribute. We set this to `application/x-shockwave-flash`, telling the browser that we are loading a Flash object. With this explicit type definition in place, Chromium browsers now use the `object-src` directive as desired. This type definition is also required in similar tests that use the `<embed>` tag.

Coverage Improvements

We could improve our coverage of the `object-src` directive by also testing that the policy is applied to objects loaded with the `<applet>` HTML element. This would require some automatic feature detection, since the `<applet>` tag is intended for loading a Java applet and therefore requires a Java runtime and browser plugin.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
</head>
<body>
  <!-- Chromium tries to load the page as a frame without the explicit type, and
       therefore looks at frame-src not object-src. We're trying to test object-
       src here, so use this type attribute to force it. -->
  <object data="/csp/fail/37" type="application/x-shockwave-flash"></object>
</body>
</html>
```

Listing 22: Testing the `object-src` CSP directive with the `<object>` tag

5.2.10 Images

We test the Content Security Policy for images – specified with the `img-src` directive – with the `<img />` HTML tag. In each test we simply try to load an image in the standard way. We could improve on this by also trying to load images via CSS and by creating the `<img />` tags in our existing tests dynamically with JavaScript.

5.2.11 Media

The `media-src` CSP directive restricts the origins from which a protected resource may load audio and video. We test this using the `<audio>` and `<video>` tags introduced in HTML5. Because these tags are not supported in all browsers, we first check that the browser supports them using *Modernizr*'s `Modernizr.audio` and `Modernizr.video` properties.

We encountered some issues with the media tests in older versions of Safari. The problem was that the load event on the frame loading the audio/video was not being fired. This resulted in our tests timing out, since the frame was never being loaded by the browser as far as the test code was concerned. We did not want to fix this with a timeout, and instead investigated how we could cause the load event to fire as soon as the audio or video had been loaded. The first thing we tried was to serve the `/csp/pass/TEST_ID` and `/csp/fail/TEST_ID` responses for the media tests with a `Content-Type` header specifying a MIME type for audio or video. This fixed the problem in Safari 6, however it persisted in Safari 5. The next thing we tried was to serve real audio and video files in the responses instead of just pretending to with the `Content-Type` header. We found two small and openly-licensed `.mp3` and `.mp4` files to serve on the relevant pass and fail URLs. In order to illustrate this more clearly, the relevant server-side Go code is shown in Listing 23.

```
// Page load event is not fired in older versions of Safari unless the audio
// and video have good Content-Type and are actually served
if 88 <= idInt && idInt <= 97 {
	http.ServeFile(w, r, "./csp/mpthreetest.mp3")
} else if 98 <= idInt && idInt <= 107 {
	http.ServeFile(w, r, "./csp/small.mp4")
}
```

Listing 23: Serving real audio and video files where necessary in order to fix Safari page load events

The disadvantage of this approach is that the audio and video responses are now much larger than they were previously. The MP3 file we use is 196 KB whilst the MP4 video file is 376 KB. These will not be cached by the browser, either: we send headers to tell the browser not to cache `/csp/pass/TEST_ID` and `/csp/fail/TEST_ID` URLs for the reasons discussed in Section 4.2.2. If the overhead caused by the audio and video files becomes a problem in the future, we could perhaps detect whether or not serving the media files is necessary. In most browsers, it is not necessary to serve them at all. In some, it is sufficient to serve only the `Content-Type` header but not the files in the request body. Safari 5 is the only browser we have found that implements the Content Security Policy that requires us to serve the media. We could either attempt to detect Safari 5 using the browser's user agent string, or instead detect whether or not the frame load event is fired when we want it to be by doing one or two preflight tests, and determining whether or not timeouts occur due to the frame load events not firing.

5.2.12 Frames

We test the CSP's `frame-src` directive by loading frames both using the `<iframe>` tag and the `<frame>` tag. In the `<frame>` tests, we use the `<frameset>` tag instead of `<body>` to ensure that the HTML is valid. We could improve on our `frame-src` tests by also testing the CSP when frames are created dynamically in JavaScript, and also by testing what happens when we try to navigate an already-existing frame to a disallowed URL.

5.3 Cross-Origin Resource Sharing

We cover the technical background for these browser tests in Section 2.2.3. There are 60 cross-origin resource sharing (CORS) tests in total, separated into the following four sections based on the CORS response header to which they relate:

- `Access-Control-Allow-Origin`;

- `Access-Control-Allow-Methods`;
- `Access-Control-Allow-Headers`;
- `Access-Control-Expose-Headers`;

There are some gaps in the coverage of these tests compared with the CORS specification. There are no tests to determine whether the browser correctly separates requests into simple and non-simple requests based on the HTTP methods and headers present, as defined in the W3C Recommendation. There are technical limitations in our application design preventing us from testing this at present. To detect whether or not a request is treated as simple or non-simple by the browser, we should be able to detect the preflight request sent by the browser in the case of a non-simple request. Recall that this request is sent with the `OPTIONS` HTTP method before the actual request is sent. We cannot detect this easily with the current application because the browser does not send any cookies with the preflight request, even if `withCredentials` is set to `true`. As such, we cannot save a flag in the session representing whether or not a preflight request was received: since no cookies are sent with the preflight request, we cannot associate this request with the user's existing session. This problem should be solvable in the future, perhaps by sending some unique key to the server in the request URL so that the server can associate the `OPTIONS` request with a user's session even in the absence of cookies[5]. These limitations also have the side-effect that it would be problematic for us to test CORS requests involving credentials, and so we do not test these at present. The final gap in the current test coverage of CORS is that we are unable to test that preflight requests are cached by the browser according to the `Access-Control-Max-Age` header.

Before any of these CORS tests are executed, we first detect browser support for CORS using *Modernizr*'s `Modernizr.cors` Boolean property. All of the CORS requests are made to `https://test.browseraudit.com`. These are cross-origin requests since the page making the requests (the test page) has origin `https://browseraudit.com`.

5.3.1 Access-Control-Allow-Origin

This category of tests is checking for correct behaviour of the main concept of CORS: a server can allow *XMLHttpRequests* to come from a different origin as long as it is specified by the `Access-Control-Allow-Origin` header. With each request going to the origin `https://test.browseraudit.com`, we check that the request is:

[5]Note that the problem described here is the same issue that prevented us from testing the `report-uri` directive of the Content Security Policy in Section 5.2 – the `POST` request containing the report is sent without cookies

- blocked when no `Access-Control-Allow-Origin` header is set (i.e. the server has not enabled CORS);
- allowed with `Access-Control-Allow-Origin: https://browseraudit.com` (as this is the origin of the page making the request);
- allowed with `Access-Control-Allow-Origin: *` (the wildcard value);
- blocked with `Access-Control-Allow-Origin:https://test.browseraudit.com` (the origin of the document being requested, and a subdomain of the host making the request, but not the same origin as the document making the request);
- blocked with `Access-Control-Allow-Origin: https://browseraudit.org` (an origin completely different to the document making the request and the document being requested).

This covers a wide selection of possible cases, including the wildcard value and different types of origin mismatches. These tests are implemented in the JavaScript functions `originExpectAllowed()` and `originExpectBlocked()`. Each of these takes two arguments: the textual test title to be displayed to the user and the value of the `Access-Control-Allow-Origin` header to be set by the server. The former function (shown in Listing 24) is called when the CORS request should be allowed, whereas the latter is called if the request should be blocked by the browser. jQuery's `$.ajax()` function is used to make the requests; we use the `success` callback to detect a request that has been allowed, and the `error` callback to detect a blocked request. Each request is made to a URL of the form `cors/allow-origin/ALLOWED_ORIGIN` where `ALLOWED_ORIGIN` is a Base64 encoding of the origins to be allowed by the server. The decoded version of this value is served by the server in a `Access-Control-Allow-Origin` header with the response to the request.

5.3.2 Access-Control-Allow-Methods

These tests ensure that the `Access-Control-Allow-Methods` response header can be used by the server to specify non-simple request methods that may be used to access a resource. This header is used to test requests made by all non-simple HTTP methods that are supported by *XMLHttpRequest*. The `TRACE`, `CONNECT` and `PATCH` methods are not tested since these are not supported as *XMLHttpRequest* methods by all major browsers. We do not test the `Access-Control-Allow-Methods` header with the simple methods (`GET`, `HEAD` and `POST`) since this does not make sense and the header will be ignored by the browser, even if the header tries to state that the simple method used is illegal.

The `PUT`, `DELETE` and `OPTIONS` HTTP methods are tested. Since these are non-simple

```
function originExpectAllowed(desc, allowOrigin) {
  it(desc, function(done) {
    $.ajax({
      url: "https://test.browseraudit.com/cors/allow-origin/"+
        $.base64.encode(allowOrigin),
      success: function() {
        done();
      },
      error: function(r, textStatus, errorThrown) {
        var moreInfo = (errorThrown !== "") ? " "+errorThrown : "";
        throw "Request "+textStatus+moreInfo;
      }
    });
  });
}
```

Listing 24: The `originExpectAllowed()` function testing the `Access-Control-Allow-Origin` CORS header

methods, the requests should be allowed only if the method is included in a `Access-Control-Allow-Methods` header sent by the server. Recall that this header can contain a comma-separated list of allowed methods. For each method tested, we have five tests. Using `PUT` as an example, the test cases are as follows:

- method is allowed with header `Access-Control-Allow-Methods: PUT` (a single allowed method that matches the actual request method);
- method is allowed with header `Access-Control-Allow-Methods: DELETE, PUT` (a list of allowed methods, one of which is the actual request method);
- method is blocked with header `Access-Control-Allow-Methods:DELETE` (a single allowed method that doesn't match the actual request method);
- method is blocked with header `Access-Control-Allow-Methods:DELETE, TRACE` (a list of allowed methods, neither of which is the actual request method);
- method is blocked when no `Access-Control-Allow-Methods` header is set.

The tests for each method (`PUT`, `DELETE` and `OPTIONS`) follow the pattern above, resulting in 15 tests in this category overall. These tests cover a good range of possible header values, ensuring that the cross-domain CORS request is always correctly allowed or blocked as expected. These tests are implemented in the `methodExpectAllowed()` and `methodExpectBlocked()` JavaScript functions. As well as the textual test description, each function takes as arguments the actual request method `requestMethod` as well as the list of allowed methods `allowedMethods` which will be set in the `Access-Control-Allow-Methods` header by the server. As in the origin tests seen already, we use jQuery's `success` and `error` AJAX callbacks to detect whether or not a CORS request has been allowed.

5.3.3 Access-Control-Allow-Headers

These tests are testing the behaviour of the `Access-Control-Allow-Headers` CORS response header. This response header can be used by the server to specify the non-simple request headers it is happy to receive. Throughout these tests, we send requests containing combinations of the custom headers `X-My-Header` and `X-Another-Header`. We test these requests with a wide range of possible `Access-Control-Allow-Headers` header values. The tests are implemented in the `headersExpectAllowed()` and `headersExpectBlocked()` functions which accept the test title, actual request headers and allowed request headers as arguments. Listing 25 shows an example call to the `headersExpectBlocked()` function used by one of the tests. We can see that the actual request headers are passed as a JavaScript object, allowing us to set as many headers as we want in the single function argument. In this case we expect the request to be blocked since the CORS request contains two custom headers yet only one of them is allowed by the `Access-Control-Allow-Headers` header.

```
headersExpectBlocked("...textual description...",
  {
    "X-My-Header": "foo",      // actual request headers
    "X-Another-Header": "bar"
  },
  "X-My-Header");              // allowed request header(s)
```

Listing 25: An example call to `headersExpectBlocked()` when testing the `Access-Control-Allow-Headers` CORS header

The `headersExpectAllowed()` and `headersExpectBlocked()` functions work in a similar way to all other CORS tests discussed so far. Each request is made to a URL of the form `cors/allow-headers/ALLOWED_HEADERS`, where the value passed in `ALLOWED_HEADERS` is used by the server in the `Access-Control-Allow-Headers` header. As always, allowed and blocked cross-origin requests are detected with jQuery's `success` and `error` callbacks from the `$.ajax()` function.

5.3.4 Access-Control-Expose-Headers

The final category of CORS tests checks for the correct behaviour of the `Access-Control-Expose-Headers` response header. This header can be used by the server to specify the non-simple response headers that the calling JavaScript may access after receiving a response from a cross-origin request. The caller can typically access a response header using the `getResponseHeader()` method on an *XMLHttpRequest*. Since we are using jQuery to simplify CORS requests, we must first gain access to the *XMLHttpRequest* object. This is easily done since it is passed as the third argument to the `success` callback. The tests for header exposure are implemented in two functions `exposeExpectAllowed()` and `exposeExpectBlocked()`. Each of these takes three arguments: `desc`, `header` and `exposedHeaders`. `header` is the header to which access is attempted; `exposedHeaders` is a comma-separated list of exposed headers that will be set by the server in an `Access-Control-Expose-Headers` header.

Simple Request Headers

We first test that access to the six simple response headers is granted even in the absence of an `Access-Control-Expose-Headers` header from the server. These headers are `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified` and `Pragma`. Not all of these headers are sent by our Nginx/Go server combination by default, and so the Go handler behind these requests has been written to always set all six of the headers. Each of these 6 tests consists of a call to `exposeExpectAllowed()` since it is always expected that access to the header will be allowed.

Non-Simple Request Headers

The more interesting `Access-Control-Expose-Headers` tests are those in which the caller attempts to access a non-simple response header with the `getResponseHeader()` method. The non-simple headers that we attempt to access are `Server`, `Content-Length`, `Connection` and `Date`. All of these are headers that are sent by default by our server, and so they are returned by the handler behind this category of CORS tests. For each header, we test that access is correctly allowed and blocked in a number of different cases. Using `Server` as an example header, we test that:

- the caller can access the `Server` header with `Access-Control-Expose-Headers: Server` (a single header is allowed which matches the header being accessed);
- the caller can access the `Server` header with `Access-Control-Expose-Headers: Content-Length, Server` (two headers are allowed, one of which is the header being accessed);
- the caller cannot access the `Server` header with `Access-Control-Expose-Headers: Content-Length` (a single header is allowed which does not match the header being accessed);

- the caller cannot access the `Server` header with `Access-Control-Expose-Headers: Content-Length, Connection` (two headers are allowed, neither of which matches the header being accessed);
- the caller cannot access the `Server` header when no `Access-Control-Expose-Headers` header is set by the server.

This pattern applies to all 4 non-simple request headers that are tested, resulting in 20 tests overall. The tests are implemented with a request to `cors/exposed-headers/EXPOSED_HEADERS` where `EXPOSED_HEADERS` is the value passed to the server that will be returned in a `Access-Control-Expose-Headers` header.

5.4 Cookies

The technical background for these tests is covered in Section 2.2.4. The tests cover the *HttpOnly* and *Secure* cookie attributes.

5.4.1 HttpOnly Flag

We have three tests related to the *HttpOnly* cookie flag. The first test is testing for the correct behaviour of an *HttpOnly* cookie set by the server – it should be inaccessible by JavaScript. The latter two tests are testing what happens if we try to create an *HttpOnly* cookie in JavaScript. It should be discarded by the browser immediately, which means that it should be inaccessible by JavaScript and should also not be sent to the server.

Cookie Set by Server

This is a single test that tests the browser's compliance with the *HttpOnly* flag when the cookie is set by the server. Listing 26 shows the source code of the test. The browser simply requests a page that sets an *HttpOnly* cookie. It then tries to access this cookie, and expects its value to be undefined. We are using the `jquery.cookie` jQuery plugin[6] to access cookies in JavaScript, which simplifies the process and avoids us having to parse `document.cookie` ourselves.

This test evolved from the proof-of-concept Mocha test shown back in Section 3.1.1. The key difference is that the proof-of-concept made two HTTP requests: one to clear any

[6] `https://github.com/carhartl/jquery-cookie`

```
it("cookie should be inaccessible by JavaScript", function(done) {
  $.get("/httponly_cookie", function() {
    expect($.cookie("httpOnlyCookie")).to.be.undefined;
    done();
  });
});
```

```
const HTTPONLY_COOKIE_NAME = "httpOnlyCookie"
const HTTPONLY_COOKIE_SETVAL = "619"

func HttpOnlyCookieHandler(w http.ResponseWriter, r *http.Request) {
  DontCache(&w)

  expires := time.Now().Add(5 * time.Minute)
  cookie := &http.Cookie{Name: HTTPONLY_COOKIE_NAME,
    Value:    HTTPONLY_COOKIE_SETVAL,
    Path:     "/",
    Expires:  expires,
    HttpOnly: true}
  http.SetCookie(w, cookie)
}
```

Listing 26: Client- and server-side code of the first *HttpOnly* cookie test

already-existing cookie value, then the second to set it as we do in Listing 26. The first request is not needed since the second request would overwrite any existing cookie anyway (since the name and scope are identical), so the delete request in the original proof-of-concept test was redundant.

Cookies Set by JavaScript

These are two tests that check the browser's behaviour when we create an *HttpOnly* cookie with JavaScript. The JavaScript source of the tests is displayed in Listing 27.

Each test creates a new cookie in JavaScript using `document.cookie`. Note that we can't use the `jquery.cookie` plugin to simplify the syntax for creating the cookies; it doesn't support creating an *HttpOnly* cookie, which makes sense because it is not something we should be able to do! The cookies are created by setting `document.cookie` to a new value that sets the cookie. This does not overwrite any existing cookies (e.g. our session cookie).

In the first test we create an *HttpOnly* cookie `discard` in JavaScript and then try to read its value. We expect it to be undefined – the browser should immediately discard the cookie since it makes no sense for JavaScript to create an *HttpOnly* cookie.

```javascript
it("cookie set by JavaScript should be discarded and inaccessible by JavaScript",
function() {
  document.cookie = "discard=browseraudit; HttpOnly";
  expect($.cookie("discard")).to.be.undefined;
});

it("cookie set by JavaScript should discarded and not be sent to server",
function(done) {
  document.cookie = "destroyMe=browseraudit; HttpOnly";
  $.get("/get_destroy_me", function(destroyMe) {
    expect(destroyMe).to.equal("nil");
    done();
  });
});
```

Listing 27: Testing that an *HttpOnly* cookie set by JavaScript is immediately discarded

The second test is similar, but instead of testing to see if we can read it in JavaScript, we test to see whether or not the `destroyMe` cookie is sent to the server. We make a request to `/get_destroy_me`, a page served by the golang server which outputs the value (if any) of the `destroyMe` cookie sent with the request. We expect this to be "nil", which is what the Go server will output if the cookie was not in the request.

5.4.2 Secure Flag

We have four tests that check that the browser correctly implements the *Secure* cookie flag. These can be viewed as two pairs of tests; in each pair we are testing that a secure cookie is sent to the server over HTTPS but not over HTTP. In the first pair of tests, the secure cookie is set by the server. In the second pair the cookie is set by JavaScript.

Secure Cookie Set by Server Sent Over HTTPS

The first test checks that a cookie set by the server with the *Secure* flag set is sent to the server over the `https:` protocol; its source is displayed in Listing 28. We can see that this test is slightly more complicated than the "cookie set by server" *HttpOnly* test that we have seen already, since it makes two requests to the server.

The first request sets the secure cookie – the JavaScript sends a request to the Go web server whose response contains a `Set-Cookie` header that sets a cookie with the *Secure* flag. We must then send a second request in order to determine whether or not the browser will send the cookie over an HTTPS connection (as it should). We make this second request to a page handled by the golang server that returns the value of the

```
it("cookie should be sent over HTTPS", function(done) {
  $.get("/secure_cookie", function() {
    var secureCookie = $.cookie("secureCookie");
    $.get("/get_request_secure_cookie", function(data) {
      expect(data).to.equal(secureCookie);
      done();
    });
  });
});
```

```
func SecureCookieHandler(w http.ResponseWriter, r *http.Request) {
        DontCache(&w)

        expires := time.Now().Add(5 * time.Minute)
        cookie := &http.Cookie{Name: SECURE_COOKIE_NAME,
                Value:   SECURE_COOKIE_SETVAL,
                Path:    "/",
                Expires: expires,
                Secure:  true}
        http.SetCookie(w, cookie)
}

func GetRequestSecureCookieHandler(w http.ResponseWriter, r *http.Request) {
        DontCache(&w)

        c, err := r.Cookie(SECURE_COOKIE_NAME)
        if err == nil {
                fmt.Fprintf(w, "%s", c.Value)
        } else {
                fmt.Fprintf(w, "nil")
        }
}
```

Listing 28: Testing that a secure cookie is sent over HTTPS

`secureCookie` cookie sent by the browser, or “nil” if it is not present. In the test we then assert that the cookie value returned by the server is equal to the value we were expecting (that is, the cookie value originally set in the first request). If this is not the case then the browser has failed the test since it did not send the secure cookie received in the first response with the second request.

Secure Cookie Set by Server Not Sent Over HTTP

This is the one of the more interesting secure cookie tests – testing that a secure cookie set by the server is *not* sent over plain HTTP. The client- and server-side for the test can be seen in Listing 29. Note that this test sends three requests to the server, not two as in the previous secure cookie test.

The first request to the server is exactly the same: the client-side makes a request to the server whose response sets the secure cookie. The second request serves the same purpose as in the first test: we want to determine whether or not the browser sends the secure cookie with this request. However, this request must be sent over plain HTTP, which complicates things. As we already know, we cannot make a cross-origin *XMLHttpRequest* due to the same-origin policy. Recall that our test page is loaded over HTTPS, so we cannot use jQuery's `$.get()` function to make a request over plain HTTP to (and receive the response from) a page that returns the cookie value sent with the AJAX request. We therefore split this step into two requests:

1. load an *image* over plain HTTP from a page that records in the server-side session the value of `secureCookie` sent with the image request;
2. send an AJAX request over HTTPS to a page that returns the cookie value stored in the session during the first request.

This implementation allows us to avoid being blocked by the same-origin policy whilst still correctly testing the browser, and is a pattern we have seen in many of our security tests discussed so far (not just those to do with cookies).

Note that the case on line 25 of the golang code in Listing 29 should never be hit. The request to `/get_session_secure_cookie` should always come after `/set_secure_cookie`, and so a cookie value should exist in the session. We ensure that this is the case in the JavaScript by using a load handler on the image we load. In order for this to work in all browsers we tested, we had to serve an image in the response for the load callback to be executed. This happens on line 14 of the Go code in Listing 29, where we serve a single pixel image.

Secure Cookie Set by JavaScript

We have two further secure cookie tests. The first tests that a secure cookie set by JavaScript is sent over HTTPS. The second, and more interesting, test is testing that a secure cookie set by JavaScript is not sent over plain HTTP. The implementation of these two tests is very similar to the two *Secure* flag tests seen so far. The only difference is that these tests set the cookie in JavaScript, and so one fewer request is required in each.

```
it("cookie should not be sent over HTTP", function(done) {
  $.get("/secure_cookie", function() {
    $("<img />", { src: "http://browseraudit.com/set_secure_cookie" }).load(function() {
      $.get("/get_session_secure_cookie", function(data) {
        expect(data).to.equal("nil");
        done();
      });
    });
  });
});
```

```
func SetSecureCookieHandler(w http.ResponseWriter, r *http.Request) {
  DontCache(&w)

  session, _ := store.Get(r, "browseraudit")

  c, err := r.Cookie(SECURE_COOKIE_NAME)
  if err == nil {
    session.Values[SECURE_COOKIE_NAME] = c.Value
  } else {
    session.Values[SECURE_COOKIE_NAME] = "nil"
  }

  session.Save(r, w)
  http.ServeFile(w, r, "./static/pixel.png")
}


func GetSessionSecureCookieHandler(w http.ResponseWriter, r *http.Request) {
  DontCache(&w)

  session, _ := store.Get(r, "browseraudit")

  c := session.Values[SECURE_COOKIE_NAME]
  if c == nil {
    log.Println("nil secure cookie")
  }

  fmt.Fprintf(w, "%s", c)
}
```

Listing 29: Testing that a secure cookie is not sent over plain HTTP

5.5 Request Headers

In this category we test browser security features related to miscellaneous HTTP request headers that do not fit into any other category. The technical background for these tests is not discussed in Chapter 2 but instead discussed alongside the implementations.

5.5.1 Referer

This is a single test to check that the `Referer` request header is not sent with a non-secure request if the referring page was transferred over a secure protocol. Put simply, if a plain HTTP request was referred by a document loaded over HTTPS, then the request should not include the `Referer` header. This behaviour is defined in RFC 2616, and exists to prevent the leaking of private information [15].

The HTTP `Referer` request header (originally a misspelling of referrer) is most commonly sent when a user clicks a link from one webpage to another. Its purpose is to allow a web server to learn where a request originated from. For example, many webmasters use the header to learn about the sources of their web traffic. The header is not only sent when a user clicks a link to another document, but also whenever a document is loaded by another document. For example, when an HTML webpage loads an image with an `<img />` tag, the request for the image's `src` is sent with a `Referer` header containing the URL of the document loading the image. We take advantage of this in our test, since it is easier for us to programmatically load an image in JavaScript than it is to simulate a link click.

Since our test page is already loaded over a secure protocol, we can simply load an image from the golang web server over plain HTTP[7]. The server records the `Referer` header (if any) sent with the image request in the user's session. The JavaScript then sends a second request, obtaining the referrer stored in the previous request. This second request is an AJAX request over HTTPS. We expect the referrer value received to be empty. The two requests that comprise this test cannot be squashed into one due to same-origin policy restrictions as seen in many other tests.

[7]This will result in a warning along the lines of "the connection to this website is not fully secure because it contains unencrypted elements (such as images)" in most browsers, usually displayed with a warning icon to the left of the URL bar

5.6 Response Headers

In this category we test browser security features related to miscellaneous HTTP response headers that do not fit into any other category.

5.6.1 X-Frame-Options

The technical background for these tests is covered in Section 2.2.6.

This category of tests checks that the browser correctly implements the anti-clickjacking header `X-Frame-Options` as documented in RFC 7034 [32]. It is worth noting that the RFC itself states that "not all browsers implement `X-Frame-Options` in exactly the same way, which can lead to unintended results". This quickly became apparent when testing this category of tests since, at the time of writing, 2 out of the 8 fail in the latest version of Chromium. The behaviour of our tests is supported both by the RFC, and by two other webpages[8,9] we found online that carry out some simple `X-Frame-Options` testing similar to our tests. It is also important to recall that today's browsers implement differing behaviour when it comes to nested frames. Because of this variation we must load our test frames directly from the page running our tests so that there are no nested frames. Nested frames lead to mixed behaviour that has not been formally defined, so we cannot define an expected result. This limits our testing scope – we cannot run any `X-Frame-Options` tests that involve nested frames because of this mixed and undefined behaviour.

At present, we only test `X-Frame-Options` with the `<iframe>` element. Current browser implementations of `X-Frame-Options` apply to much more than just the `<iframe>` tag, covering a wider range of tags that enable HTML content from other domains: `<frame>`, `<object>`, `<applet>` and `<embed>` tags. We test whether or not our frame has been rendered by having an image inside the framed document and detecting whether or not the request for the image reached the server side. This is a similar pattern to the one seen in our Content Security Policy tests (see Section 5.2).

All `X-Frame-Options` tests use the same JavaScript function `frameOptionsTest()` on the client-side. This function is displayed in Listing 30. Its four arguments are as follows:

- `desc` (string) – the textual test description passed to Mocha's `it()` function;
- `shouldBeBlocked` (Boolean) – `true` if and only if we expect the loading of the frame to be blocked due to the `X-Frame-Options` header;

[8]`http://erlend.oftedal.no/blog/tools/xframeoptions/`
[9]`http://www.enhanceie.com/test/clickjack/`

- `sourcePrefix` (string) – the prefix appended to the frame's URL, specifying the origin from which it should be loaded, e.g. `https://test.browseraudit.com`;

- `frameOptions` (string) – the value of the `X-Frame-Options` header to be sent with the framed document.

```javascript
function frameOptionsTest(desc, shouldBeBlocked, sourcePrefix, frameOptions) {
  frameOptionsTest.id = frameOptionsTest.id || 0;

  var id = frameOptionsTest.id++;
  it(desc, function(done) {
    var defaultResult = (shouldBeBlocked) ? "pass" : "fail";
    var frameOptionsBase64 = $.base64.encode(frameOptions);

    $("<iframe>", { src: sourcePrefix+"/frameoptions/"+id+"/"+defaultResult+"/"+
      frameOptionsBase64 })
    .css("visibility", "hidden").appendTo("body").load(function() {
      $.get("/frameoptions/result/"+id, function(result) {
        expect(result).to.equal("pass");
        done();
      });
    });
  });
}
```

Listing 30: `frameOptionsTest()` function used for all `X-Frame-Options` tests

The `frameOptionsTest()` function allows us to dynamically create and serve a framed document from a specified origin with a specified `X-Frame-Options` header. This means that each of the 8 tests under this category is written in a single line of JavaScript – a call to this function. We do not need to add anything on the server-side each time an `X-Frame-Options` test is added.

The function executes a test as follows:

- a unique test ID is generated. Test IDs are sequential integers starting at 0;

- a frame is loaded from the server. This request makes use of the `sourcePrefix` argument (so that the frame is loaded from the correct origin) and also passes to the server the test ID, a default result (pass or fail) and a Base64 encoding[10] of the `X-Frame-Options` header value;

- on the server-side, the default result is stored in the session as the result for the `X-Frame-Options` test with the given ID;

[10] We use Base64 because the source prefix may contain characters with special meaning in a URL, e.g. `/`, which would break the route parser on the server-side. Passing the source prefix in the URL encoded in Base64 easily avoids any such problems, and can be decoded quickly and easily by the Go server

- the server serves the frame with the given `X-Frame-Options` header. The content of this frame is generated dynamically from a template; the frame loads an image from a URL that, if loaded, will update the result in the session for that test ID to be the opposite of the default result stored previously;

- once the frame load event is triggered (i.e. it has actually been loaded, or the browser has blocked it), the client-side queries the server for the result of the test. The server responds with the result stored in the session for that test ID.

The template from which the frame's content is generated is shown in Listing 31. The result and test ID are populated by the golang web server using the data passed in the frame request by lines 9–10 of `frameOptionsTest()`. We can see how the loading of an image is used to detect whether or not the browser has rendered a frame. Depending on the test, the frame being rendered may result in a test passing or failing, hence the result is a variable. A call to `/frameoptions/pass/4` updates the test result for the `X-Frame-Options` test with ID 4 to be a pass, for example. This is the result that is later queried in the JavaScript once the browser reports that the frame has been loaded.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
</head>
<body>
  <img src="/frameoptions/{{.Result}}/{{.TestId}}" alt="" />
</body>
</html>
```

Listing 31: The template used to generate the frame's content in an `X-Frame-Options` test

As already mentioned, we have 8 tests in this category:

- frame from the same origin with `DENY` (should be blocked);

- frame from the same origin with `SAMEORIGIN` (should be allowed);

- frame from the same origin with `ALLOW-FROM browseraudit.com` (should be allowed);

- frame from the same origin with `ALLOW-FROM test.browseraudit.com` (should be blocked);

- frame from the remote origin with `DENY` (should be blocked);

- frame from the remote origin with `SAMEORIGIN` (should be blocked);

- frame from the remote origin with `ALLOW-FROM test.browseraudit.com` (should be blocked);
- frame from the remote origin with `ALLOW-FROM browseraudit.com` (should be allowed).

The "same origin" in the tests above refers to `https://browseraudit.com`, i.e. the same origin as the test page that is loading the frames. The "remote origin" is `https://test.browseraudit.com`. Since our test page is loaded over HTTPS, we can only load frames over HTTPS as well due to the mixed-content rules. This means we are unable to test more origin mismatches that involve different schemes.

5.6.2 Strict-Transport-Security

We have five browser tests related to HTTP Strict Transport Security (HSTS) and the `Strict-Transport-Security` header. The technical background for these tests is discussed in Section 2.2.5.

In each of the HSTS tests, we clear the HSTS state once the test is complete. This is so that HSTS does not wrongly affect any future tests. We achieve this using one of Mocha's nice features: `afterEach()`. This allows us to register a function to be called upon completion of each HSTS test by placing a call to `afterEach()` inside the HSTS test category's `describe()` block. Clearing the HSTS state after each test with `afterEach()` is beneficial in two ways. Firstly, it keeps the code tidier by reducing duplication. Secondly, and most importantly, it ensures that the HSTS state is cleared even if a test fails. If we had alternatively tried to clear the HSTS state at the bottom of each test, before the `done()` call, this would not have been executed in the event of a test failure. This is because the execution of a test stops as soon as an assertion fails. Using `afterEach()` ensures that HSTS is cleared even in the event of a test failure. This is crucial, so that the failure of one test does not affect any subsequent tests. HSTS is cleared by requesting a document from the server that responds with the header `Strict-Transport-Security: max-age=0`.

Further Requests Not Sent Over Plain HTTP

This test is testing the key functionality of HSTS by ensuring that, once HSTS has been enabled by the server, further requests will not be sent over plain HTTP but instead rewritten to use HTTPS. Most of the source code is displayed in Listing 32; the handler for `/get_protocol`, which simply returns the protocol value stored in the session, has been omitted.

Recall that requests reaching the golang web server (and thus being processed by the handlers in Listing 32) are being proxied by our Nginx web server. This has the unfortunate side-effect that the requests to the Go web server do not include the full request URI. For example, a request to `http://browseraudit.com/set_protocol` looks to the golang server like a request to `/set_protocol`. Consequently, the golang server cannot learn from the HTTP request whether the request was sent over the `http:` or `https:` protocol. To work around this, we configure the Nginx server to add a custom `X-Scheme` header when proxying a request for `/set_protocol`. This is easily done with a `proxy_set_header` directive, taking advantage of the `$scheme` Nginx variable.

Now that this has been explained, the test itself should be fairly easy to follow since it follows a similar pattern to the tests seen already. The client-side first makes a request to the server whose response enables HSTS for 10 seconds. We then make a second request to the server – a plain HTTP request, which we hope will be rewritten to use HTTPS. This request is to `/set_protocol`, which records the protocol used to access it in the session. The browser then sends a third request (using AJAX, so that we can read the response data) to learn from the server the protocol recorded during the previous request. We expect this protocol to be "https"; if it is not then the plain HTTP request was not rewritten by the browser.

Doesn't Apply to Subdomains with includeSubdomains Omitted

This test ensures that HSTS does not apply to subdomains if the `Strict-Transport-Security` header does not contain the `includeSubdomains` flag. The test works much like the first HSTS test, except the request to `http://browseraudit.com/set_protocol` is instead sent to `http://test.browseraudit.com/set_protocol`. In the assertion line, we expect the protocol to equal "http" since the request should not have been rewritten.

Does Apply to Subdomains with includeSubdomains Present

This tests works exactly like the previous test, except the `Strict-Transport-Security` header *is* sent with the `includeSubdomains` flag. We therefore expect the protocol to equal "https" as the request to `http://test.browseraudit.com/set_protocol` should have been rewritten by the browser.

Ignored if Set Over Plain HTTP

In this test we are checking that the browser ignores a `Strict-Transport-Security` header if it is delivered over plain HTTP. We create an image to send a request to `http://browseraudit.com/set_hsts`, as opposed to using an AJAX call to `/set_hsts` over HTTPS as in all other HSTS tests. We cannot use AJAX for this because the request would be blocked due to it being cross-origin. This explains why we serve the `pixel.png` image in Listing 32 even though we have only been requesting it so far with an AJAX request and not looking at the received data: we need our `.load()` event to work, and so have to serve the image. We could have used a different handler that does not serve the image for the tests using AJAX, saving bandwidth, but decided not to as this would unnecessarily complicate the server-side code.

After the browser has received the `Strict-Transport-Security` header sent over plain HTTP, the rest of the test continues as normal and expects the received protocol to be "http". The request to `/set_protocol` should not be rewritten since the HSTS header sent over plain HTTP should be ignored by the browser.

Expires after max-age

This last test checks that the HSTS state expires after the number of seconds given by `max-age`. Its client-side source code is displayed in Listing 33. The first thing to note is that, in line 2, we reset the Mocha timeout for this test. Since this is a slow test, Mocha would otherwise terminate the test before its completion. Recall from the server-side handler for `/set_protocol` in Listing 32 that HSTS is set to last for 10 seconds. In this test we make a request to `/set_protocol`, wait for 15 seconds using JavaScript's `setTimeout()`, and then test to see whether our request to a plain HTTP URL is rewritten by the browser. We expect that it will not be, since the browser's HSTS state should have expired, and so we expect the protocol received from the server to be "http".

```
it("browser should not send further requests over plain HTTP", function(done) {
  $.get("/set_hsts", function() {
    $("<img />", { src: "http://browseraudit.com/set_protocol" }).load(function() {
      $.get("/get_protocol", function(protocol) {
        expect(protocol).to.equal("https");
        done();
      });
    });
  });
});
```

```
const PROTOCOL_KEY = "protocol"

func SetHSTSHandler(w http.ResponseWriter, r *http.Request) {
  DontCache(&w)

  w.Header().Set("Strict-Transport-Security", "max-age=10") // 10 seconds
  http.ServeFile(w, r, "./static/pixel.png")
}

func SetProtocolHandler(w http.ResponseWriter, r *http.Request) {
  DontCache(&w)

  session, _ := store.Get(r, "browseraudit")

  if r.Header["X-Scheme"][0] == "http" || r.Header["X-Scheme"][0] == "https" {
    session.Values[PROTOCOL_KEY] = r.Header["X-Scheme"][0]
    session.Save(r, w)
  } else {
    log.Printf("Unrecognised protocol %s", r.Header["X-Scheme"][0])
  }

  http.ServeFile(w, r, "./static/pixel.png")
}
```

Listing 32: Testing that plain HTTP requests are rewritten to HTTPS when HSTS is enabled

```
it("should expire after max-age", function(done) {
  this.timeout(16000); // 16 seconds

  $.get("/set_hsts", function() {
    setTimeout(function() {
      $("<img />", { src: "http://browseraudit.com/set_protocol" }).load(function() {
        $.get("/get_protocol", function(protocol) {
          expect(protocol).to.equal("http");
          done();
        });
      });
    }, 15000); // 15 seconds (HSTS should last for 10)
  });
});
```

Listing 33: Testing that HSTS expires after `max-age` seconds

6 Evaluation

Our project is doing very well and has captured the state of the art. There is nothing else like BrowserAudit publicly available – we have produced a web application capable of automatically providing a detailed report on a browser's implementations of a good selection of security features.

It is very difficult to find security bugs in official releases of browsers. Entire teams are dedicated to assessing the security of a browser throughout its development, with the hope that no bugs will exist by the time it is released. Browsers are also built over years and years, with each version improving on a previous one. Because of this, one would expect most bugs to have been ironed out over time. Mozilla Firefox has a selection of unit tests related to browser security that aim to automatically test the same features that BrowserAudit does, but at the C++ level (Firefox's implementation language) rather than with a web application written primarily in JavaScript. We expect that other browsers also have similar practices, although it is difficult to know for sure with the closed-source browsers. Bearing all of this in mind, it is important to remember that BrowserAudit can be used in browsers that are not official releases. For example, it could be used by a developer who is in the process of developing the next version of a web browser. BrowserAudit could also be useful for somebody who has made a custom modification to a browser but wants to ensure that its security features are still intact. Our project is not just intended for use by end users to assess their browser security – we hope that it will prove to be very useful to those involved in any form of browser development.

6.1 Mozilla Firefox Bugs

Despite our comments above about it being difficult to find security bugs in official browser releases, BrowserAudit detects two bugs in the latest (version 29.0) release of Mozilla Firefox. This proves the usefulness of our application – it automatically identifies security bugs in one of today's most popular browsers! Both bugs are in Firefox's implementation of the Content Security Policy (CSP): four of our CSP tests fail in Firefox. These bugs also exist in all prior versions of Firefox that implemented the CSP. After some further investigation to ensure that the problem was definitely Firefox and not our tests, we reported each of these bugs to Mozilla, and they were both promptly accepted as valid bugs.

6.1.1 CSP allows local CSS @import with only 'unsafe-inline' set

The Bugzilla report for this bug can be found at `https://bugzilla.mozilla.org/show_bug.cgi?id=1007205`.

We found a way to load a CSS stylesheet despite a policy that should block the stylesheet from being loaded. Consider the HTML in Listing 34 when served with the header `Content-Security-Policy: default-src 'none'; style-src 'unsafe-inline'`.

```
1 <style>
2   @import url("/should_be_blocked.css");
3 </style>
```

Listing 34: HTML used to load a stylesheet that should be blocked by the CSP

The `Content-Security-Policy` header first sets a `default-src 'none'` directive, stating that no resources may be loaded. We then specify the `style-src 'unsafe-inline'` directive, meaning that style rules are allowed inside `<style>` tags and `style="..."` attributes.

In the latest version of Firefox, line 2 from Listing 34 is allowed to load the local stylesheet `/should_be_blocked.css`. This is incorrect behaviour. The CSP header is based around whitelisted sources, and the source `/should_be_blocked.css` is not whitelisted by any of the directives in the header. If, for example, we had specified `default-src 'self'` or `style-src 'unsafe-inline' 'self'` then we would expect the stylesheet to be loaded.

This incorrect behaviour only seems to occur in Firefox when the stylesheet being loaded is a local one, i.e. from the same origin as the HTML file being served. When we replace `/should_be_blocked.css` with a remote stylesheet, it is correctly blocked. This is verified by other BrowserAudit tests that pass in Firefox.

6.1.2 CSP allows local Worker construction with only 'unsafe-inline' set

We reported this bug on Bugzilla at `https://bugzilla.mozilla.org/show_bug.cgi?id=1007634`.

The bug is very similar to the CSS import bug, except that this time we are able to load a JavaScript script that should be blocked by the CSP, using a `Worker` (or `SharedWorker`) constructor. Consider the HTML shown in Listing 35 being served with `Content-Security-Policy: default-src 'none'; script-src 'unsafe-inline'`.

```
<script>
  var worker = new Worker("/should_be_blocked.js");
</script>
```

Listing 35: HTML used to load a JavaScript file that should be blocked by the CSP

Just like the first Firefox bug, the `/should_be_blocked.js` script is wrongly loaded by the browser – there is no directive in the `Content-Security-Policy` header to allow it. When we change the `script-src` directive to `'unsafe-inline' 'self'`, the script is correctly loaded and a `Worker` constructed from it. This behaviour is confirmed by a different BrowserAudit test that passes in Firefox. As with the first Firefox bug, this bug appears only to occur when the script being loaded is from the same origin as the protected document. Remote scripts cannot be loaded in this way, which is confirmed by other BrowserAudit tests that pass in the affected versions of Firefox.

6.2 Limitations

In this section we discuss the limitations of BrowserAudit in its current state. We offer potential future improvements and fixes for some of these, but we also believe that some of the limitations cannot be worked around due to the nature of the application.

6.2.1 Tests Using Timeouts

In Section 4.3.1 we discussed the merits of using JavaScript callback functions in favour of timeouts whenever possible. Unfortunately, there are some tests for which we have been unable to find a working cross-browser implementation that does not require a timeout. This problem typically stems from an `onload` event not being fired when we require it to be. Our solution for this at the moment is to wait for a predefined amount of time, and then assume that the document (and any resources that it loads) have been loaded. The consequence of this is that our use of timeouts could potentially lead to inaccurate test results, especially in the case of a slow connection or network latency. On the other hand, we cannot simply set the timeout to a large value since this would drastically increase the time taken for all of our tests to execute.

All of the tests currently using timeouts are Content Security Policy tests whose implementations are discussed in Section 5.2. They can loosely be separated into four categories, discussed below. There were once many more cases in which load events were not fired when we desired, especially in Safari. We were able to fix these by modifying the `Content-Type` response header sent by the server with `/csp/pass/TEST_ID` and `/csp/fail/TEST_ID` responses. For example, if a CSP test loads an image using `<img`

`src="/csp/pass/42" alt="" />`, it might be beneficial to modify the `Content-Type` header for that URL to be `image/png`. In other cases, setting the `Content-Type` header was not sufficient. For example, in the CSP audio and video tests, we actually serve small audio and video files in the pass and fail responses so that the frame load event fires when we want it to (where the frame is the page loading the audio or video).

Worker and SharedWorker

The CSP tests involving the `Worker` and `SharedWorker` APIs use code like shown in Listing 36 to construct a web worker. In this case, the CSP should allow the worker to be constructed, i.e. `/csp/pass/46` is an allowed script source. The HTML document in the listing is loaded inside a frame by our testing page. We would like the testing code to use a load event on this frame, however this does not achieve what we want. We want the load event to be fired after the browser has made the request to `/csp/pass/46` to construct the worker on line 6. What actually happens is that the load event is fired as soon as the browser has rendered the HTML, but before it loads the worker. This results in our test code querying the server for the test result (which will have already been set to "fail" as the default result) before it has been set to "pass" by the worker construction on line 6. This results in a test result erroneously being reported as a failure when the browser has actually behaved correctly as far as the CSP is concerned. The only problem is the order of the requests – our test code queries the test result before the worker construction sets the test to have passed.

```html
 1  <!DOCTYPE html>
 2  <html lang="en">
 3  <head>
 4    <meta charset="utf-8" />
 5    <script>
 6      new Worker("/csp/pass/46");
 7    </script>
 8  </head>
 9  <body>
10  </body>
11  </html>
```

Listing 36: Constructing a worker in a CSP test

The load event behaviour is like this in all browsers we have found that support the `Worker` and `SharedWorker` APIs. The behaviour makes sense since workers are intended to run asynchronously in the background, in parallel with their main page. The behaviour is not ideal for us, however, and so we instead use a timeout. The test code that produced the frame waits an amount of time (currently 300ms) after which we assume that the worker construction will have been attempted.

setTimeout() and setInterval()

For the CSP `'unsafe-eval'` tests that try to use `setTimeout()` and `setInterval()` to execute a string as code in a similar manner to `eval()`, using a load event on the frame is not an option. This is because the JavaScript executed in the framed document (see Listing 37) sets a timeout itself. The code on the test page that created the frame must wait until after the `setTimeout()` timeout fires and the redirect occurs before it queries the server for the test result. A load event on the frame would fire as soon as the HTML document had been rendered, not after the timeout. This is also the case with `setInterval()`, and is the case in all browsers. Our solution is to set a timeout in the testing code longer than the 100ms timeout set by `setTimeout()` and `setInterval()` before requesting the test result from the server.

```
setTimeout('window.location = "/csp/pass/15"', 100);
```

Listing 37: Setting a 100ms timeout with `setTimeout()`

Object and Embed Tags

This problem appears to be specific to Chromium-based browsers. If we frame an HTML document that loads another resource with either an `<object>` or `<embed>` tag, the frame's load event fires before the request to load the object is made by the browser. This affects 20 of our CSP tests, all of which make use of timeouts as described previously to avoid the problem.

Inline Event Handlers

In two of our CSP `'unsafe-inline'` tests, we use a `<body onload='...'>` event to test inline event handlers. This onload event redirects the frame to a URL that sets the test result on the server. As in the other cases we have seen already, the frame's load event will be fired before the inline event that makes the redirect. We use timeouts to work around this and (hopefully) ensure that we only query the server for the test result after the inline event and redirect have occurred.

Towards a Solution: postMessage

postMessage is an API that allows messages to be sent between two windows or frames across domains. This is a secure way of sending messages cross-origin without being blocked by the same-origin policy. The sender specifies the origins of the recipients allowed to receive the message, and the recipient is given the origin of the sender as well as the message itself.

Rather than relying on load handlers on frames in our CSP tests, we could potentially use the *postMessage* API to have the frame send its parent a message, letting it know that it has tried to load the resource protected by the CSP. The JavaScript test would wait to receive this message before querying the server for the test result. The *postMessage* API is supported in all major browsers, so browser support is not a concern. The main concern with this idea is that, when a browser blocks a resource from being loaded due to the CSP, it throws a security exception. This is problematic since it can result in the browser refusing to execute a script any further. We recently had the idea to use two separate `<script>` tags which read and write a global variable to work around this. This is a possible solution for many of the cases in which we currently use timeouts, which we hope to experiment with in the future.

6.2.2 Incomplete Test Coverage

In many of our browser security tests we only test cross-origin mismatches where the two origins differ in host. There are multiple cases in which we neglect to test origins that differ in scheme or port when we perhaps should, since this leads to incomplete test coverage of the features for which we have written tests.

We are unaware of any technical limitations that would stop us from testing cross-origin port mismatches. There are technical limitations when testing origins that differ in scheme, however, due to mixed content rules. We discussed the restrictions imposed by mixed content rules in the relevant sections of Chapter 5. Since our test page is loaded over HTTPS, the test page cannot frame a document loaded over HTTP. This means that we cannot run many cross-origin tests involving the unencrypted `http:` scheme. We can think of two possible solutions to this, which we describe briefly below.

The first solution is to use popup windows: the HTTPS test page can create a popup window for an HTTP document which can then carry out the test and close itself. This could lead to a poor user experience, although the popup windows could perhaps be hidden behind the main test page with the `blur()` method.

Our second proposed solution is to load two separate test pages, one over HTTP and the other HTTPS, and aggregate the results into a single results page. The two test pages could be loaded in parallel, or we could ask the user to run one batch of tests and then the second. Having a test page loaded over plain HTTP would make it simple for us to test cross-origin tests involving the `http:` scheme. We cannot simply run all tests from a plain HTTP test page, though, since many of the test implementations are much more trivial due to the test page being loaded over HTTPS (e.g. the HTTP Strict Transport

Security tests). By having two test pages, we could implement a wider range of tests and run each test on whichever page would result in a simpler implementation.

6.3 Browser Support

Browser support is important for a project such as this. After all, it is likely that older browsers are those that will contain security bugs or lack certain security features. It is difficult to write JavaScript web-applications that work reliably across all browsers. When discussing the implementations of our browser tests in Chapter 5, we often described various techniques we use to ensure correct cross-browser behaviour. There are two key aspects to browser support that we look for when testing BrowserAudit in a new browser: correct core functionality (e.g. the tests run automatically to completion, and the result counts and progress bar are updated) and the accuracy of the test results displayed.

Given the technical challenges faced throughout this project, we are overall happy with BrowserAudit's browser support although there is definitely room for improvement. In this section we briefly discuss the results of running BrowserAudit in both the latest and historical versions of some of today's most popular browsers.

6.3.1 Mozilla Firefox

BrowserAudit runs well in all versions of Mozilla Firefox tested: both older versions and the latest builds. The tests always run to completion and the results reported are accurate based on the security features we know that Firefox supports. We tested two desktop versions of Firefox and also the Android version.

Desktop Version 29.0 (64-bit Linux)

BrowserAudit reports 304 passed tests and 6 failed tests in Firefox 29.0 on Linux. These results are accurate – the 6 failed tests are the Content Security Policy tests identifying bugs discovered in the browser, as discussed in Section 6.1.

Desktop Version 26.0 (64-bit Linux)

The BrowserAudit results in Firefox 26.0 on Linux are very similar to those in version 29.0. The application reports 302 passes and 4 failures. Note that 4 fewer tests are executed in the older version. This is because the older version supports the `Worker` API but not the `SharedWorker` API. Our Content Security Policy tests detect this and so do not run the `SharedWorker` tests. The other failures are the same as in version 29.0, identifying the security bugs found by our application.

Android Version 30.0

We tested BrowserAudit in the Android mobile version of Firefox as well as the two desktop versions. The outcome was mostly positive – the application runs well, reporting 276 passed tests and 14 failed. Whilst most test results seem to be accurate, there are some problems as highlighted by the relatively high number of failed tests. The Content Security Policy tests involving the `<audio>` and `<video>` HTML5 tags show failed results. This is not due to a security bug in the browser, but instead a difference in how Firefox for Android handles audio and video compared to the desktop versions. Most likely for data optimisation reasons, the browser does not send a request to the server for audio or video content until after the user presses the "play" button. Our tests rely on the request for the media content reaching the server without any user interaction, so that we can test the security policy automatically. In fact, the user cannot even see the audio or video to click play because it is contained within a hidden frame! We either need to determine a method of forcing the browser to load the media (perhaps by automatically playing it), or we could instead choose not to run media tests in mobile browsers.

6.3.2 Internet Explorer

BrowserAudit does not run well in any versions of Internet Explorer other than the latest version, Internet Explorer 11. Fortunately, Internet Explorer 11 has been the most popular version of Internet Explorer since January 2014 [38]. We were able to fix many problems in the older versions (the application originally didn't run at all!) but some still exist. BrowserAudit will run in Internet Explorer versions 9+ but with issues. It fails to run at all in versions 8 and older, largely because the Chai assertion library we use extensively only supports IE 9+.

Internet Explorer 10 (64-bit Windows)

When accessed in Internet Explorer 10, BrowserAudit runs and reports 232 test passes and 74 failures. This failure count is high, partially due to errors in the tests rather than security flaws in the browser. Tests fail in the Content Security Policy, `X-Frame-Options` and `Strict-Transport-Security` categories.

We expect many Content Security Policy (CSP) tests to fail since Internet Explorer (even in the latest version) does not implement the policy using the header defined in the specification. That said, the CSP tests fail in IE 10 due to Mocha *timeouts*, not our assertion failures. The `X-Frame-Options` and `Strict-Transport-Security` tests also fail due to timeouts in the same way. These timeouts are occurring due to `<iframe>` load events not being fired. This is a problem encountered multiple times throughout our implementation, for example with the media CSP tests discussed in Section 5.2.11. In all other browsers, we were able to utilise various response headers to make the load event be fired when we need it to be. We have been unable to achieve this in IE 10.

Internet Explorer 11 (32-bit Windows)

The load event issues in IE 10 do not exist in Internet Explorer 11. BrowserAudit works very well in IE 11, reporting 222 passed tests and 64 failures. There are no timeout issues as in IE 10. The tests that fail are in the Content Security Policy, `X-Frame-Options` and `Strict-Transport-Security` categories. Coincidentally, many of these tests are the same tests that fail in IE 10. The key difference is that the tests fail properly in IE 11 as opposed to in IE 10. These features are not implemented in Internet Explorer according to the specification and so BrowserAudit flags this with its failed tests.

6.3.3 Safari

In many sections of the Browser Tests chapter (Chapter 5) we described a wide range of tweaks implemented purely to fix the tests in Apple's Safari browser. Thanks to these tweaks, BrowserAudit works well in most versions of Safari. The application runs correctly in all versions tested (5–7), although there are still some timeout issues in Safari 5.

Desktop Version 6

In version 6 of the desktop version of Safari, BrowserAudit reports 280 passes and 6 failures in total. The application runs well (and very quickly!) and all failures reported are accurate. 2 of the failures are due to lack of support for the `ALLOW-FROM` values of `X-Frame-Options`, just as we saw in Google Chrome for Android. The other 4 failures occur because Safari 6 does not support HTTP Strict Transport Security.

iOS (iPhone and iPad)

We also tested BrowserAudit in the mobile versions of Safari on both an iPhone and an iPad. It runs in these browsers just as well as in the desktop version of Safari. The results are exactly the same: 280 passes and 6 failures.

6.3.4 Android Stock Browser

When explaining the motivation behind this project back in Section 1.1, we remarked that many deployments of the Android stock browser lack various important security features, yet are still used in practice by so many users today. Unfortunately we do not have access to an Android 2.3 device (the version referenced in our introduction that doesn't even implement *HttpOnly* cookies) but we have been able to test the stock browser of two more recent Android versions. The application runs very well in both versions.

Android 4.2.2

Running on the stock browser of Android 4.2, BrowserAudit reports 203 passed tests and 64 failed tests. Most of these failures are accurate. 47 of them are due to the browser failing to implement the Content Security Policy. A further 3 of the failures are a result of the browser not implementing HTTP Strict Transport Security. The more interesting failed tests are part of the Cross-Origin Resource Sharing section. Some tests fail in Android 4.2's stock browser because the `getResponseHeader()` method (see Section 2.2.3) cannot be used to retrieve any non-simple response header, even when allowed by the server. We are undecided on whether this is a problem in the browser's CORS implementation or instead a problem in the *XMLHttpRequest* implementation. The `getResponseHeader()` method works as-expected when accessing simple request headers (which are not protected by CORS). The remaining failures are not entirely accurate and are instead due to problems with the tests. The tests for the `X-Frame-Options` header fail due to timeouts as in other browsers discussed already. This is down to the browser not executing load event callbacks when we need it to.

Android 4.4.2 (Google Chrome)

The stock browser in Android 4.4 is a mobile version of Google Chrome. BrowserAudit runs brilliantly in this, with excellent results. The application reports 284 test passes and just 2 failures. The only failing tests are due to Chrome not implementing the `ALLOW-FROM` values of the `X-Frame-Options` header. This is also the case in desktop versions of Chrome and in other browsers such as Safari. We acknowledged that not all browsers support this value when discussing the background of the feature in Section 2.2.6.

6.4 User Experience Feedback

We visited the departmental "Project Fair" to harness some feedback on the interface of our application. Recall that when describing the interface design in Section 3.1.2, we stressed the importance of our application's interface being appropriate for a wide range of users. We hope that BrowserAudit is usable by all kinds of users ranging from non-technical users to browser developers. The majority of the feedback came from technical users who are not knowledgeable about web security. We also demonstrated the application to some more typical web users to ascertain that the application is usable by (and useful for) a non-technical user.

The feedback we received was overall very positive. The technical users especially found the project to be useful and interesting. The website design was well-received by all, with multiple users commenting on the usefulness of the progress bar and "Show/Hide Details" button. One user commented on how the website looks good on multiple devices, even those with low screen resolutions. Some of the negative comments received about the user experience were as follows:

- when expanding or collapsing a category of test results, the entire bar should be clickable rather than just the title text;
- it is not clear when the tests have completed, especially on small screens;
- one user commented that they thought all web browsers are already secure.

We can act on this feedback in the future to make the website even more accessible than it already is.

7 Conclusions

We have produced a web application that automatically tests the implementations of many standard browser security features in the browser used to access our website. BrowserAudit tests a wide range of security features, both those that are crucial and should be implemented by all browsers, and more recent security features that will only be implemented by modern browsers but are becoming increasingly important to prevent new kinds of attacks.

All of our tests run automatically with no input required from the user. This is what makes BrowserAudit unique compared to other works we have been able to find – there is no other publicly-accessible web application that automatically tests so many aspects of a web browser's security like ours does. BrowserAudit's interface has been designed with all kinds of users in mind. An average web user can run BrowserAudit's tests and gain a simple assessment on his browser – critical, warnings or okay. A more technical user can view a detailed breakdown of each test result, and see which security features passed our tests and which had problems. If he is not quite sure what a browser security feature is, he can learn about it (and how we test it) thanks to the textual descriptions we have written for each category of tests. For a highly advanced user, we provide exact details of why a test has failed, including actual and expected results where possible. This kind of information should be useful for a browser developer or security researcher.

We were not necessarily expecting to detect any bugs in any modern browser. We instead thought that the most interesting results would come from running BrowserAudit in older browsers that are still commonly used. That said, BrowserAudit did expose two new bugs in the latest version of Mozilla Firefox.

We have intentionally developed BrowserAudit in a way such that new tests can be written and added in a simple manner. In a way, it can be viewed as the beginning of a framework for writing browser tests. We explain the importance of this when discussing possible future work in Section 7.1.

7.1 Future Work

In this section we discuss various improvements that could be made to the BrowserAudit project in the future. Many of these are things that we would have liked to implement if more time had been available to work on the project. That said, we plan to continue working on the project over the coming months, and so we may implement many of the below items shortly.

7.1.1 Testing More Security Features

We currently test several of the most important browser security features, including the primary aspects of the same-origin policy which we believe to be the most important concept in browser security. There are many more security features that we should be able to automatically test on BrowserAudit. Some of these features fit nicely inside the test categories that can be seen on the application today, whereas others would lead to a brand new category of browser tests. The more features BrowserAudit can test, the better. Some ideas that we have for future BrowserAudit tests are described below.

Same-Origin Policy

In the background for the same-origin policy in Section 2.2.1, we stated 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.

postMessage

In Section 6.2.1 we briefly discussed the *postMessage* API for sending messages between two different windows across domains without being blocked by the same-origin policy. We proposed the use of this API as a possible solution for problems in our implementation in which we are forced to use timeouts where would prefer to avoid them.

postMessage is an API that would be good to test on BrowserAudit. It is used by many developers to avoid the headaches sometimes inflicted by the same-origin policy. 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. *postMessage* is also something that is tested by *Browserscope*, a related project covered in Section 2.3.1. *Browserscope* only tests for support of the API though, rather than thoroughly testing its implementation from a security viewpoint.

X-Content-Type-Options

`X-Frame-Options` is a security-related HTTP response header first introduced in Internet Explorer 8 [23]. It is now also supported by Google Chrome and Safari, whilst the Firefox team is still debating its implementation [8].

The header has just one valid value: `nosniff`. It is designed to prevent a browser from *MIME-sniffing* a response away from its declared `Content-Type`. This prevents attacks based around MIME type confusion. MIME-sniffing is a method used by browsers to try to work out the a document's real MIME type by looking at the content itself, instead of using the document's `Content-Type` header. To explain what this means, Listing 38 shows the example HTTP response given in Microsoft's initial explanation. The `X-Content-Type-Options` header is on line 5. The key behaviour of the header is explained in the text on line 9: in a browser that supports the header, the response body will be rendered as plaintext. The browser will display the source HTML in the window – it will not render it. In a browser that doesn't support the header (e.g. Mozilla Firefox), the page will likely be rendered as HTML despite the `Content-Type` header on line 4 stating that the content type of the document is plaintext (`text/plain`). We could write BrowserAudit tests to test the browser's support for the `X-Frame-Options` header.

```
1  HTTP/1.1 200 OK
2  Content-Length: 108
3  Date: Thu, 26 Jun 2008 22:06:28 GMT
4  Content-Type: text/plain
5  X-Content-Type-Options: nosniff
6
7  <html>
8  <body bgcolor="#AA0000">
9  This page renders as HTML source code (text) in IE8.
10 </body>
11 </html>
```

Listing 38: Microsoft's example of the `X-Content-Type-Options` header

Heartbleed as a client

The Heartbleed bug (`CVE-2014-0160`) is a serious vulnerability recently discovered in the popular OpenSSL library. The bug existed for over two years before being publicly disclosed. Most of the online reports about the bug refer to the use of OpenSSL by web servers, and how the bug could be exploited to expose private data from the server's memory, including sensitive information such as private keys and user passwords.

The bug actually works both ways – client software could also be vulnerable if connecting to an evil or compromised server. Many refer to this as "reverse heartbleed". The bug could be exploited by a server to expose data on the client computer. As we have seen already, web browsers act as TLS clients when loading webpages over HTTPS. Any browser built with a vulnerable OpenSSL version could be exploitable, and this is something that can be tested. Fortunately, none of the major five browsers uses the OpenSSL library. This might not be the case for all browsers, though, and so we could test whether the user's browser is vulnerable to reverse heartbleed when acting as a TLS client. Automatic testing for reverse heartbleed is not a new idea, and has been implemented by others already[1].

'How's My SSL?' TLS Results

When discussing *How's My SSL?* as related work in Section 2.3.2, we suggested that we could make use of their tests in our project. A JSON API is provided, including all of the information that we would need to present a BrowserAudit user with the test results from *How's My SSL?* regarding their browser as a TLS client. Since *How's My SSL?* is hosted on a different domain (and therefore a different origin) to BrowserAudit, however, we cannot make an AJAX request from BrowserAudit for this JSON document. If we wanted to include their results in BrowserAudit, we would have to run our own copy of *How's My SSL?* (which is open source and conveniently written in Go) on the `browseraudit.com` domain. This could be worthwhile, since the results would be very interesting and are especially relevant to browser security. Alternatively, we could contact the creator of the site and ask whether he would add a cross-origin resource sharing (CORS) header (see Section 2.2.3) allowing BrowserAudit to request the JSON API with the *XMLHttpRequest* API. This would only work in browsers that implement CORS.

7.1.2 Better Coverage of Features Already Tested

At many points in this report, we have suggested potential future improvements relating to the test coverage of features for which we already have tests. These improvements are discussed in each detail as the implementation of each section of tests is discussed, however a brief list of potential coverage improvements is as follows:

- in many tests involving origin mismatches, we only test origins that differ in host rather than scheme or port. This is primarily due to technical limitations discussed

[1]`https://reverseheartbleed.com/`

as a limitation in Section 6.2.2 where we also propose some solutions. We could explore one of these solutions to improve our test coverage;

- there are some features of the Content Security Policy that we do not test, such as the `font-src` and `connect-src` directives, sandboxing, and the reporting features. We also don't test any CSP directives where a resource is loaded from a URL that redirects;

- recall that a page should not be able to set a cookie's scope to a top-level domain (TLD). For example, `subdomain.example.com` can set a cookie's scope to `*.example.com` but not to `*.com` since this is too broad. This leads to an interesting case with country-code TLDs (ccTLDs). For example, `waw.pl` should be seen as a TLD, and so it should not be possible to set a cookie's scope to be this broad. It should, however, be possible to set a cookie's *Domain* parameter to `example.pl`. There are many ccTLDs, and this can be problematic for browser developers. If we were to purchase domains using ccTLDs, we could test browser implementations of this case.

7.1.3 Improving Browser Support

We found it very difficult to find a single method to automatically test a feature that works across as many browsers as possible. In Section 6.3 we discussed the current state of browser support in BrowserAudit, commenting on how there is definitely room for improvement. We hope that we will be able to improve on the browser support of already-existing BrowserAudit tests, paying special attention to Internet Explorer versions 9 and 10 which are not the latest version but still have a reasonable usage share.

7.1.4 Open Sourcing BrowserAudit

After writing some developer documentation and making some small tweaks, we hope to open source the BrowserAudit project, probably on GitHub. We will then invite other developers to expand the BrowserAudit codebase by writing more tests. This will hopefully demonstrate the advantages of us writing the codebase in a way such that tests can easily be added in a modular fashion. Other developers will be able to write their own tests (or improve/fix already-existing tests) and then make a pull request. We would then review the changes and merge them into the BrowserAudit website, so that their tests are ran by anyone who accesses BrowserAudit. Through open sourcing the project, we believe that BrowserAudit has the potential to grow into a popular hub for browser security testing that developers can both contribute to and benefit from.

7.1.5 Automatic Testing

When discussing how we test our security tests, we suggested in Section 3.3.1 that we could use a tool called Selenium WebDriver to automatically test our web application. After the initial work of learning how to use the tool and writing tests in Java, WebDriver would allow us to easily and automatically test that BrowserAudit behaves as expected in a wide range of browsers. In order to write the WebDriver tests, we would first need to document exactly which tests we expect to have okay, warning and critical statuses in each browser. This could take some time, but we still believe that this could be worth experimenting with in the future.

7.1.6 Feature Ideas

Beyond improving BrowserAudit purely by adding and improving browser tests, we also have some ideas for new features of the web application that could be added in the future. These ideas are described briefly below.

Ability to run a subset of tests

At present, it is only possible to run our full set of 300+ browser tests. If a user were only interested in the result of a specific test, or perhaps a category of tests (e.g. Content Security Policy), he has no choice but to run all of the tests and wait for the test result(s) he is interested in to be displayed. It is very feasible that a user may want to repeatedly run only a subset of our tests, for example when investigating a particular test that fails.

We could implement a solution that allows a user to select a specific subset of tests to run. We could possibly even make use of Mocha for this, which uses a query string parameter `grep` to run only those tests that match the given string. For example, visiting the URL `https://browseraudit.com/test?grep=cookie` runs only our tests that match the string "cookie". We do not document this feature on BrowserAudit, and it is not currently substantial enough to restrict the tests as well as a user might want. For example, setting the string to "Same-Origin Policy" still runs tests outside of the Same-Origin Policy category.

A BrowserAudit API

We could extend BrowserAudit to provide an API for web developers. This API would allow a developer to query our API, providing us with the user agent string of his visitor. Our API would then respond with information about the BrowserAudit test results for the visitor's browser. The web developer could program his application such that it then deploys the best security setup given the security features he knows that the visitor's browser supports thanks to BrowserAudit's API.

Sharing of test results

It would be useful if a user could share his BrowserAudit test results with somebody else. Upon test completion we could provide the user with a URL that can be used by another visitor to view the test results. This shared test results page could also include information such as the date and time that the tests were ran and details of the browser used (e.g. Firefox 29.0 on Ubuntu, and any extensions installed) as well as the test results themselves.

Storing and publishing test results

One nice feature of *Browserscope*, a related work discussed in Section 2.3.1, is their summaries of browser results. This is pictured in Figure 2.4 on page 30. *Browserscope* detects the browser being used to access their website and updates their database of test results for that browser once they are complete. This results in an accurate picture of their test results across all browsers that have been used to access the page. This would be nice to have on BrowserAudit. In the same way that *BrowserScope* can be used to gain a good idea of which browsers implement various features, BrowserAudit could become a good source of information on browser-wide security implementations.

Bibliography

[1] Kunal Anand. *Go (programming language): Is Google Go ready for production use?* 21st May 2012. URL: `https://www.quora.com/Go-programming-language/Is-Google-Go-ready-for-production-use/answer/Kunal-Anand?share` (visited on 09/06/2014).

[2] A. Barth. *HTTP State Management Mechanism.* RFC 6265 (Proposed Standard). Internet Engineering Task Force, Apr. 2011. URL: `http://www.ietf.org/rfc/rfc6265.txt`.

[3] A. Barth. *The Web Origin Concept.* RFC 6454 (Proposed Standard). Internet Engineering Task Force, Dec. 2011. URL: `http://www.ietf.org/rfc/rfc6454.txt`.

[4] Adam Barth and M West. 'Content Security Policy 1.1'. In: *W3C Working Draft WD-CSP11-20130604* (2013).

[5] T. Berners-Lee, R. Fielding and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0.* RFC 1945 (Informational). Internet Engineering Task Force, May 1996. URL: `http://www.ietf.org/rfc/rfc1945.txt`.

[6] T. Berners-Lee, R. Fielding and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax.* RFC 3986 (INTERNET STANDARD). Updated by RFC 6874. Internet Engineering Task Force, Jan. 2005. URL: `http://www.ietf.org/rfc/rfc3986.txt`.

[7] Peter Bourgon. *Go at Soundcloud.* 24th July 2012. URL: `https://developers.soundcloud.com/blog/go-at-soundcloud` (visited on 09/06/2014).

[8] *Bug 471020 – Add X-Content-Type-Options: nosniff support to Firefox.* 8th Mar. 2014. URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=471020` (visited on 09/06/2014).

[9] *Can I use Content Security Policy.* URL: `http://caniuse.com/contentsecuritypolicy` (visited on 20/05/2014).

[10] *Can I use Cross-Origin Resource Sharing.* URL: `http://caniuse.com/cors` (visited on 21/05/2014).

[11] *Chromium Embedded Framework.* Valve Developer Community. URL: `https://developer.valvesoftware.com/wiki/Chromium_Embedded_Framework` (visited on 19/02/2014).

[12] *Dashboard – Android Developers. Platform Versions.* Google Inc. URL: `https://developer.android.com/about/dashboards/index.html` (visited on 19/02/2014).

[13] Robert Duncan. *SSL: Intercepted today, decrypted tomorrow.* Netcraft Ltd. 25th June 2013. URL: http://news.netcraft.com/archives/2013/06/25/ssl-intercepted-today-decrypted-tomorrow.html (visited on 02/06/2014).

[14] Peter Eckersley. 'How unique is your web browser?' In: *Privacy Enhancing Technologies.* Springer. 2010, pp. 1–18.

[15] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Draft Standard). Updated by RFCs 2817, 5785, 6266, 6585. Internet Engineering Task Force, June 1999. URL: http://www.ietf.org/rfc/rfc2616.txt.

[16] Josh Fraser. *The security hole I found on Amazon.com.* 6th June 2014. URL: http://www.onlineaspect.com/2014/06/06/clickjacking-amazon-com/ (visited on 11/06/2014).

[17] Ian Hickson and David Hyatt. 'HTML5: A vocabulary and associated APIs for HTML and XHTML'. In: *W3C Candidate Recommendation CR-html5-20140429* (29th Apr. 2014). URL: http://www.w3.org/TR/2014/CR-html5-20140429/browsers.html#security-window (visited on 09/06/2014).

[18] J. Hodges, C. Jackson and A. Barth. *HTTP Strict Transport Security (HSTS).* RFC 6797 (Proposed Standard). Internet Engineering Task Force, Nov. 2012. URL: http://www.ietf.org/rfc/rfc6797.txt.

[19] *Home – BrowserScope. Android 2.3.* BrowserScope. URL: http://www.browserscope.org/?category=security&v=3&ua=Android%202.3* (visited on 19/02/2014).

[20] Monsur Hossain. *Using CORS.* HTML5 Rocks. 26th Oct. 2011. URL: http://www.html5rocks.com/en/tutorials/cors/ (visited on 16/06/2014).

[21] *HTTP access control (CORS). Requests with credentials.* Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS#Requests_with_credentials (visited on 14/06/2014).

[22] *HttpOnly – OWASP. Browsers Supporting HttpOnly.* The Open Web Application Security Project. URL: https://www.owasp.org/index.php/HttpOnly#Browsers_Supporting_HttpOnly (visited on 19/02/2014).

[23] *IE8 Security Part VI: Beta 2 Update. MIME-Handling: Sniffing Opt-Out.* 2nd Sept. 2008. URL: http://blogs.msdn.com/b/ie/archive/2008/09/02/ie8-security-part-vi-beta-2-update.aspx (visited on 09/06/2014).

[24] Blake Mizerany Keith Rarick. *Go at Heroku.* 21st Apr. 2011. URL: http://blog.golang.org/go-at-heroku (visited on 09/06/2014).

[25] Jason Kincaid. *Google's Go: A New Programming Language That's Python Meets C++.* 10th Nov. 2009. URL: http://techcrunch.com/2009/11/10/google-go-language/ (visited on 07/06/2014).

[26] D. Kristol and L. Montulli. *HTTP State Management Mechanism.* RFC 2965 (Historic). Obsoleted by RFC 6265. Internet Engineering Task Force, Oct. 2000. URL: http://www.ietf.org/rfc/rfc2965.txt.

[27] Petar Maymounkov. *The Go Circuit Project is open source: An "AppEngine" for systems-level distributed engineering.* 23rd Apr. 2013. URL: https://groups.google.com/forum/?fromgroups=#!topic/golang-nuts/qelU5Lrq-uA (visited on 09/06/2014).

[28] *Netscape and Sun Announce JavaScript, the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet.* Netscape Communications Corporation. 4th Dec. 1995. URL: http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html (visited on 29/01/2014).

[29] Rick Olson. *Key/value logs in Go.* 2nd Nov. 2013. URL: http://techno-weenie.net/2013/11/2/key-value-logs-in-go/ (visited on 09/06/2014).

[30] *Persistent Client State – HTTP Cookies.* Tech. rep. Netscape Communications Corporation. URL: http://curl.haxx.se/rfc/cookie_spec.html (visited on 01/06/2014).

[31] E. Rescorla. *HTTP Over TLS.* RFC 2818 (Informational). Updated by RFC 5785. Internet Engineering Task Force, May 2000. URL: http://www.ietf.org/rfc/rfc2818.txt.

[32] D. Ross and T. Gondrom. *HTTP Header Field X-Frame-Options.* RFC 7034 (Informational). Internet Engineering Task Force, Oct. 2013. URL: http://www.ietf.org/rfc/rfc7034.txt.

[33] Jesse Ruderman. *Same-origin policy.* Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (visited on 01/06/2014).

[34] Chris Shiflett. *Twitter Don't Click Exploit.* 12th Feb. 2009. URL: http://shiflett.org/blog/2009/feb/twitter-dont-click-exploit (visited on 27/05/2014).

[35] *StatCounter Global Stats.* Jan. 2014. URL: http://gs.statcounter.com/#desktop-browser-ww-monthly-200807-201401 (visited on 13/06/2014).

[36] Brandon Sterne and Adam Barth. 'Content Security Policy 1.0'. In: *W3C Candidate Recommendation CR-CSP-20121115* (2012).

[37] Biz Stone. *Clickjacking Blocked.* Twitter Inc. 12th Feb. 2009. URL: https://blog.twitter.com/2009/clickjacking-blocked (visited on 27/05/2014).

[38] *The Internet Explorer Browser.* W3Schools. URL: http://www.w3schools.com/browsers/browsers_explorer.asp (visited on 16/06/2014).

[39] Anne Van Kesteren. 'Cross-origin Resource Sharing'. In: (16th Jan. 2014). URL: http://www.w3.org/TR/2014/REC-cors-20140116/.

[40] Michal Zalewski. *Browser security handbook.* 2010. URL: http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_DOM_access.

[41] Michal Zalewski. *HTTP cookies, or how not to design protocols.* 28th Oct. 2010. URL: http://lcamtuf.blogspot.co.uk/2010/10/http-cookies-or-how-not-to-design.html (visited on 11/06/2014).

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