BrowserAudit - Reloaded: Towards a complete framework for testing browser client-side security

Houssem El Fekih

May 23, 2015

Abstract

Contents

List of Figures

List of Tables

0.1 Introduction

0.1.1 motivation

In recent years, there has been an explosion of thick client web architecture web applications, which means a shift in the importance of this attack surface available to hackers that was previously overlooked or deemed less crutial than server logic or other serious network and server exploits.

When asked about Cross Site Scripting or Cross Site Request Forgery a lot of developers still regard these types of attacks not of paramount importance, even if they long surpassed the number of reported intrusions compared to SQL injections or other network attacks or attacks on server which are obviously of concern.

Besides developer negligence, the needs of the dynamic and vibrant web with HTML5 dominating has meant that browser vendors that delivered the specification the quickest has often had the edge, at the detriment of security considerations especially in the client side.

The Same Origin Policy was the first policy to mitigate security risks but it is dated and too restrictive to many web applications that might host features on subdomains or exhibiting inter-dependance on other web applications which is more and more common, as in mashups [4] for example.

We will need to clarify the history of features and their support on browsers along with the potential problems entailed by the progressive roll-out of standards like CORS, CSP which are still a work-in-progress due to the breadth of web technologies and evolving needs. While these mechanisms are meant to stop or make more difficult a class of attacks we arguably deal with a more serious type of defect in browser technology. Improper implementation of the standards.

BrowserAudit is a tool developed at Imperial College as part of a previous student's outstanding thesis, It has the express purpose of verifying integrity of security policies in browser, it's an aggregation of cutting-edge security considerations on the client-side. It is a useful tool for browser developers to along their unit tests for the implementation of those policies, and web developers to test their browsers It also contains experimental tests from proposed standards and hence is a good place to consider the direction web standards should be going towards in terms of securing the client-side.

0.1.2 objectives

The purpose of this paper is to briefly explain the work of this tool and to build up on it in the direction of completeness and modularity the goal is to aleviate the tension between extensibility, completeness and usability, to know for a fact when using different browsers due to thorough empirical testing the security of these browsers .

We also try to add a pedagogical aspect that ensures maximum usability and hence adoption by developers and security concientious users. Ultimately this means providing a risk analysis and expert knowlege breakdown that is informed by the data we progressively and systematically analyse by testing all the browsers in the wild.

At it's current state BrowserAudit is the starting point and cover some substantial ground already thanks to the work of c. Hothersall , and dr Sergio maffeis. For the sake of clarity we lay the natural background information in a similar but different way than in [7] we will discuss the selected features and extensions proposed future work section of that very paper, and precisely state our own proposals and the rationale behind them.

0.1.3 contribution

0.1.4 Structure

The remainder of this report consists of:

- A General background section briefly explaining the web technologies involved in detail bla
- A review of browseraudit as it stands before the takeover for this thesis and discussing the direction we decided the research to take.
- Design and Implementation general design and implementation details.
- Tests technicalities where we go more in depth of the specific tests we added through our new interface.
- our Stretch goal: risk analysis for the developer and net
- Evaluation
- Conclusion

Chapter 1

Background

1.1 History

The Web has evolved from a collection of static documents connected by hyperlinks into a dynamic, rich, interactive experience driven by client-side code and aggregation by Web services. The security policy of modern browsers was designed to avoid vulnerabilities in old sites, rather than to provide the best abstractions for the newest sites. In this section, we summarize the existing access control policies and the limitations they place on Web site design.

1.1.1 90's to 2010

The Cookie is a fundemental part of client side sessions, crutial to give users certain types of functionality on the web, they were first developed by John Giannandrea, Montulli as an attempt to implement a shopping cart website for the initial Netscape cookie specification in 94'. Version 0.9beta of Mosaic Netscape, released on October 13, 1994, supported cookies.

The first use of cookies (out of the labs) was checking whether visitors to the Netscape website had already visited the site. Support for cookies was integrated in Internet Explorer in version 2, released in October 1995.

The concept of same-origin policy **??** dates back to Netscape Navigator 2 in 1995. All modern browsers implement some form of the Same-Origin Policy as it is an important security cornerstone. are often extended to define roughly compatible security boundaries for other web technologies, such as Microsoft Silverlight, Adobe Flash, or Adobe Acrobat, or for mechanisms other than direct DOM manipulation, such as XMLHttpRequest.

A script can access its document origin's remote data store using the XMLHttpRequest object, which issues an asynchronous HTTP request to the remote server.

XML-HttpRequest is the cornerstone of the AJAX programming, and the birthplace of web 2.0 The concept behind the XMLHttpRequest object was originally created by the developers of Outlook Web Access (by Microsoft) for Microsoft Exchange Server 2000. The Mozilla project developed and implemented an interface called nsIXMLHttpRequest into the Gecko layout engine. This interface was modeled to work as closely to Microsoft's IXMLHTTPRequest interface as possible. Mozilla created a wrapper to use this interface through a JavaScript object which they called XMLHttpRequest.The XMLHttpRequest object was accessible as early as Gecko version 0.6 released on December 6 of 2000, but it was not completely functional until as late as version 1.0 of Gecko released on June 5, 2002.
The XMLHttpRequest object became a de facto standard in other major web clients, implemented in Safari 1.2 released in February 2004, Konqueror , Opera 8.0 released in April 2005 and iCab 3.0b352 released in September 2005. This is a typical example of Time-to-standard for the web and how features can become standard from a single entity shipping it, this is also the case for cookies mentioned earlier.

The World Wide Web Consortium published a Working Draft specification for the XMLHttpRequest object on April 5, 2006, edited by Anne van Kesteren of Opera Software and Dean Jackson of W3C.[17] Its goal is "to document a minimum set of interoperable features based on existing implementations, allowing Web developers to use these features without platform-specific code. " The last revision to the XMLHttpRequest object specification was on November 19 of 2009, being a last call working draft.

Microsoft added the XMLHttpRequest object identifier to its scripting languages in Internet Explorer 7.0 released in October 2006. With the advent of cross-browser JavaScript libraries such as jQuery and the Prototype JavaScript Framework, developers can invoke XMLHttpRequest functionality without coding directly to the API. Prototype provides an asynchronous requester object called Ajax.Request that wraps the browser's underlying implementation and provides access to it. jQuery objects represent or wrap elements from the current client-side DOM. They all have a .load() method that takes a URI parameter and makes an XMLHttpRequest to that URI, then by default places any returned HTML into the HTML element represented by the jQuery object.

The W3C has since published another Working Draft specification for the XMLHttpRequest object, "XMLHttpRequest Level 2", on February 25 of 2008. Level 2 consists of extended functionality to the XMLHttpRequest

object, including, but not limited to, progress events, support for cross-site requests, and the handling of byte streams.The latest revision of the XMLHttpRequest Level 2 specification is that of 16 August 2011, which is still a working draft.

1.1.2 2010-2015

As of 5 December 2011, XMLHttpRequest version 2 has been merged into the main XMLHttpRequest specification, and there is no longer a version 1 and a version 2.

Of course initially AJAX had to also follow the SOP, today's browser abstractions offer an all-or-nothing trust model for Web programmers. Site a.com either does not trust Site b.com 's content at all by segregating b.com 's content into a frame or a.com trusts b.com 's scripts entirely by embedding b.com 's scripts and giving them full access to a.com 's resources.

In order to provide more fine grained handling of access origin inlight of web 2.0 application the rigidness of SOP was aleviated through the Cross Origin Resource Sharing Policy. Proposed by Matt Oshry, Brad Porter, and Michael Bodell of Tellme Networks in March 2004 for inclusion in VoiceXML 2.1 to allow safe cross-origin data requests by VoiceXML browsers. The mechanism was deemed general in nature and not specific to VoiceXML and was subsequently separated into an implementation NOTE. The WebApps Working Group of the W3C with participation from the major browser vendors began to formalize the NOTE into a W3C Working Draft on track toward formal W3C Recommendation status.

The Content Security Policy is a more recent developement that sought to provide web designers or server administrators with much more fine grained control over how content interacts on their web sites.It helps mitigate and detect types of attacks such as XSS and data injection more directly . CSP is not intended to be a main line of defense, but rather one of the many layers of security that can be employed to help secure a web site.

at the time BrowserAudit was made the latest draft of CSP1.1 released in June 2014 was the latest and was not an official Recommendation or RFC. We have now reached Level 2 policy as of 19'th of February,[3] this is a candidate Recommendation.

Note the attachement to precise terminology regarding the specification of the main subjects of interest of this paper. This is because this paper is largely an integration project that requires an in-depth insight of the direction of the standards in order to be able to prioritise features to be added to BrowserAudit2.0

1.2 High level overview of browser functions

In this section we will go through an intuitive view of how a vanilla browser would function.

The browser communicates with the server by making HTTP requests. we will briefly describe HTTP 1.1 after going through the request format which is the browser convention this format based on URLs is later converted appropriately to HTTP requests and we will see that the particulars affect security behaviour of the browser.

1.2.1 Uniform Resource Locators

A uniform resource locator (URL) identifes a specifc resource on a remote server. Com- monly referred to as a web address, it is usually displayed prominently in a web browser's user interface. The URL syntax is detailed in [2]; 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 specifed, 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 identifer, also optional, specifes an exact location within the document. In HTML documents, these fragment IDs are often combined with anchor tags to allow hyperlinks to specifc sections within a document. The most important elements of a URL as far as we are concerned are the scheme, host and port. We have mentioned them but will see later in more details that, together as a tuple, they form a concept known as an origin used in many browser security concepts.

1.2.2 What is HTTP/1.1?

FROM THE RFC:

```
HTTP 1.1

   The Hypertext Transfer Protocol (HTTP) is an application-level
   protocol for distributed, collaborative, hypermedia information
   systems. It is a generic, stateless, protocol which can be used for
   many tasks beyond its use for hypertext, such as name servers and
   distributed object management systems, through extension of its
   request methods, error codes and headers. A feature of HTTP is
   the typing and negotiation of data representation, allowing systems
   to be built independently of the data being transferred.
   HTTP has been in use by the World-Wide Web global information
   initiative since 1990. This specification defines the protocol
   referred to as "HTTP/1.1", and is an update to RFC 2068.
```

Version 1.1 adds a couple of features compared to 1.0 regarding connection type like the chunked and keep open options. It also has better caching support with vary and cache-control and etags .There is a new HTTP method OPTIONS which is typically used for CORS. few new status codes better compression and authentication and other network and security improvement and is a lot more extensible.

1.2.3 what about HTTPS/TLS?

This is nothing more than HTTP/1.1 tunnelled through a TLS socket which is the improvement on SSL this means that traffic is much more safe from eavesdropping on the line, otherwise called man-in-the-middle attacks and means integrity properties can be met as well. How good the crypto implementation is a direction we could take but we're not too certain at the moment we will just give this a passing mention.

1.2.4 Basic interaction, request and response

The HTTP verbs we care about the most for this paper are the GET and POST methods, these have near identical syntax for requests and slight differences in responses, and the proper semantics is supposed to be that GET is used to fetch content and POST for initiating action on the server, perhaps also meaning fetching content, although with AJAX this has changed slightly

they roughly do the same thing, except GET requests are more cacheable, bookmarkeable etc.. and is easier to tamper with, there is a natural ease to use POST requests for uploading or dealing with forms because that is the most commonly held use of it and the browser developer intent behind it. Two other distinctions to keep in mind between these two HTTP verbs is that in GET requests parameters are serialised in the parameter and hence have a much smaller length (7607 characters) whereas POST requests passes the parameters in the body of the requests making the limit of 8Mb, and offers more marshalling formats. This have repercussions on the security of GET requests and in particular for opening the attack tree for CSRF attacks see **??**. an example request header:

```
GET / HTTP/1.1
Host: www.duckduckgo.com/
Connection: close
User-Agent: Web-sniffer/1.1.0 (+http://web-sniffer.net/)
Accept-Encoding: gzip[CRLF]
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Cache-Control: no-cache[CRLF]
Accept-Language: de,en;q=0.7,en-us;q=0.3
Referer: http://web-sniffer.net/
```

and the corresponding response header:

```
 -- response --
200 OK
Server:  nginx
Date:  Tue, 24 Feb 2015 21:23:21 GMT
Content-Type:  text/html; charset=UTF-8
Expires:  Tue, 24 Feb 2015 21:23:20 GMT
Cache-Control:  no-cache
Strict-Transport-Security:  max-age=31536000
Content-Encoding:  gzip
X-Firefox-Spdy:  3.1
```

The two have similar structure which is a column seperated name value pair, these are called HTTP fields and they allow to negotiate the parameters for acceptable encoding and compression of data back and forth among other things like caching, cookies etc.. The request is followed by a response of course, except when a http HEAD request is made, often to test if a resource or website is live.

Header name	Description	Example	Status
Cookie	An HTTP cookie previously sent by the server with Set-Cookie (below)	Cookie:$Version=1; Skin=new;	permanent
Origin	Initiates a request for cross-origin resource sharing (asks server for an 'Access-Control-Allow-Origin' response field) .	Origin: http://www.example-social-network.com	Permanent: standard
Referer [sic]	This is the address of the previous web page from which a link to the currently requested page was followed.	Referer: http://web-sniffer.net/	Permanent
User-Agent	The user agent string of the user agent	User-Agent: Mozilla5.0 (X11; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/21.0	Permanent

Table 1.1: List of request headers of interest to this paper.

1.2.5 Important header verbs and fields

the most important request header are the first line, Host and User-Agent.. For our purposes it is also important to look at the Refer header, which is problematic for privacy but also allows to warn against wandering off a trusted HTTPS connection ..

The tables in 1.1, 1.2 present the main headers we're interested in with a brief description for HTTP requests and responses correspondingly.

Cookies

Cookies are set through responses by the Set-Cookie command, and has always been the only way to get state in a stateless protocol such as http Cookies are sent with every subsequent request to the same domain. This meant that through the years developers added attributes to cookies to enhance security. the HTTPonly attribute mandates to the browser is only accesible through the Http request and not through the client side, making it impossible to steal the cookie directly after an XSS attack. There is also the Secure attribute, which ensures that the cookie is only included if HTTPS is

used.

Cookies are set by the server through set-cookie headers that look a little like this:

```
Set-cookie: name = value; [(attribute [= value];)*]
```

and as we said sent in subsequent requests but there is a bit more to it:

- the cookies are scoped by the origin of the server.
- Cookie origin:domain,path.
- Cookie is identified by name and origin.
- Cookie scope is determined by origin and secure attribute.
- Browser request sends all cookies that are in scope to server.
- The attributes are not sent back in the requests so server cannot know if cookie is httpOnly, set in Subdomain or by javascript.
- SOP allows elements in the same path to see cookies (as illustrating in the next Origins section).
- Even secure attribute does not guarantee cookie integrite , as we see in XST section 1.3.3

1.2.6 Origins

An origin is defined as a combination of URI scheme, hostname, and port number. For examples ,all of the following resources have the same origin: http://example.com/ http://example.com:80/ http://example.com/path/file

Each of the URIs has the same scheme, host, and port components.

Each of the following resources has a different origin from the others.

http://example.com/ http://example.com:8080/ http://www.example.com/ https://example.com:80/ https://example.com/ http://example.org/ http://ietf.org/

In each case, at least one of the scheme, host, and port component will differ from the others in the list.

Origins are required to implemet Same Origin Policy and as explained is the most rudimentary client side security mechanism.

1.2.7 the DOM, javascript and CSS

The Document Object Model The document object model is a specially connected tree of DOM elements , representing the structure of the HTML page which has an xml like semantics. Different types of tags have different display features on the browser and different default styling and animation behaviour. The DOM also allows the creation of script tags that point to scripts to be fetched and executed in the scope of the document as well as stylesheet nodes. Modern browsers add many utility and other things on top of the DOM.

Javascript Javascript is the scripting language that is shipped with browsers nowadays, It is a weakly typed dynamic functional programming language with prototypical inheritence, it was developed by Brendan Eich in 11 days at Netscape. Stressing the point that in the web often things get made very quickly to match competition or attract market and these things are sometimes not temporary. which might be a problem for security.

Javascript is in it's 6th iteration though and glaring security issues can be avoided if enough care is taken Javascript is passed to the browser in UTF-16 format and is immediately parsed and code starts to run asap usually hooking into the document.load or document.ready event and there are many libraries that support module pattern and asynchronous loading used nowadays

The quick parsing and the permissive semantics of javascript means there are possibilities of serious security flaws, the most prominent of which relate to the ability to evaluate scripts inside scripts which is heavily discouraged but has limited valid applications.

Javascript is technically called ECMAScript because the European Computer manufacturers association because they helped standardise it and settle disputes between it and microsoft's version of it: JScript that it needed to run on Internet explorer to avoid trademark issues.

CSS

Cascading stylesheets are a way of providing styling directives to the browser and is composed of the set of attributes exposed explicitely though the 'style' attribute of a DOM element and a selector language that can have classes , identifiers that are also DOM based and pseudo-selectors which are CSS specific. There are also possibilities of leakage due to the fact that the browser has to parse these and apply the values to the document sometimes not in the most secure way for the user.

1.2.8 XMLHTTPRequest

this API allows to make requests to any server after a page have been loaded from a server. As explained lengthily in the History section. this is the underlying primitive of web 2.0 applications. Which loads resources on the fly usually from many domains and is typically invoked using an AJAX call in Jquery or other basic library that programmers use.

for example this code that saves some code to server and notifies the user of the response when complete.

```
$.ajax({
type: "POST",
url: "some.php",
data: { name: "John", location: "Boston" }
})
.done(function( msg ) {
alert( "Data Saved: " + msg );
});
```

We will restate here that this is the main driver for Cross Origin Resource Sharing policy and opens up the attack surface drastically. because requests can be made sometimes undetected after a page is loaded when a script is injected or made to run somehow with the right Origin.

1.2.9 external objects

In order to have extra functionality and circumvent limitations of the browser, we have to rely on external objects that can be bundled through browser Object objects or other special tags, notable examples of these is Flash Objects or Java applets

1.3 Attacks on the client

1.3.1 XSS

Cross site scripting attacks refers to any way of getting javascript or other code running on your browser within the domain originating from the server you're accessing such that it's not in violation of the same origin policy. This essentially means that they abuse the trust you have for a particular website in order to run malicious code as if originating from the site This can be done in a number of ways, there are three main classes of XSS attacks that span

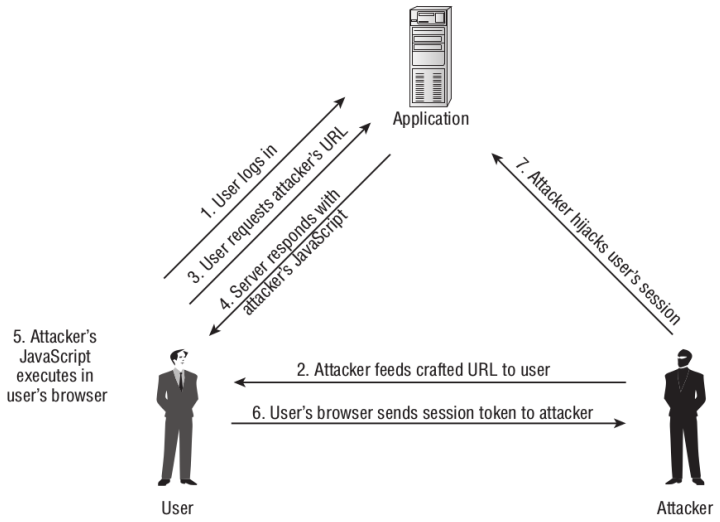

Figure 12-3: The steps involved in a reflected XSS attack

Figure 1.1: steps involved in reflected XSS attack

the possibilities.

Reflected XSS attacks

These account for 75% of XSS attacks also called first order XSS they work by abusing the fact that in certain websites developers create a unified error message function which prints back the error from a field in the URL This allows the malicious user to input a payload in an error page by supplying the URL to the user somehow and have the payload executed. The more general working can be seen in 1.1. Of course the payload can do further things like steal the cookie etc..

Stored xss attacks

In the stored variant of an XSS attack the principle is similar, but the mechanism of getting the code to run relies on any part of the website that stores arbitrary content without proper sanitasation and then renders it to users on the site this is a very common threat is social media sites or other side with aggregation of user generated content.
see 1.2 this is also called a second order XSS attack. They can be in-band meaning generated through some input on the web application itself. or out of band meaning through some other path to the database or backend holding the data.

DOM based XSS attacks

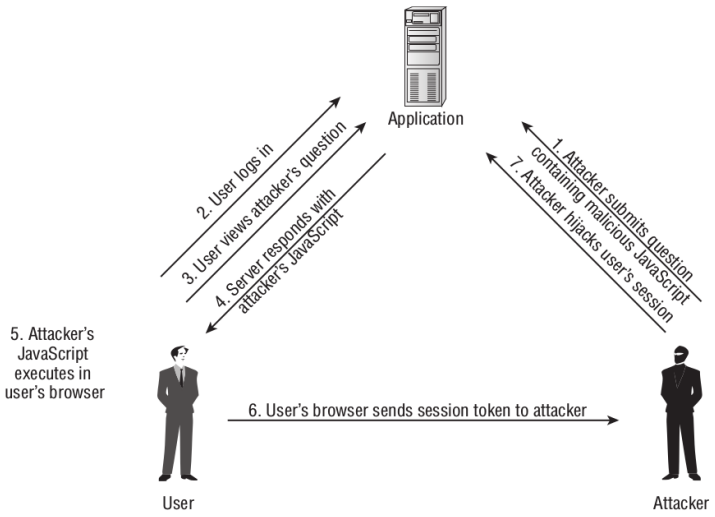

Figure 12-4: The steps involved in a stored XSS attack

Figure 1.2: steps involved in stored XSS attack

The DOM based variant is similar to the reflected XSS bug in that it requires the user to visit a crafted url. the difference is that it relies on server processing or already existing Javascript code that automatically runs that turns the crafted url into malicious javascript that gets loaded. As you can imagine this means that the attacker conducts a phase of careful investigation of the server processing of url content as well as scripts that are loaded with the page by default.

brief discussion of XSS

Reflected and stored XSS have two important differences in the attack process. Stored XSS generally is more serious from a security perspective.

First, in the case of reflected XSS, to exploit a vulnerability, the attacker must induce victims to visit his crafted URL. In the case of stored XSS, this requirement is avoided. Having deployed his attack within the application, the attacker simply needs to wait for victims to browse to the page or function that has been compromised. Usually this is a regular page of the application that normal users will access of their own accord.

Beyond the scope of preventing 'attacks to other users' and potentially nasty consequences for the web applications in terms of gaining admin. privileges and performing other attacks but another topical issue entailed by this attack vectors is stealing data. There has been a rise in the appetite for accurate data and techniques such as preventing cookies have had very

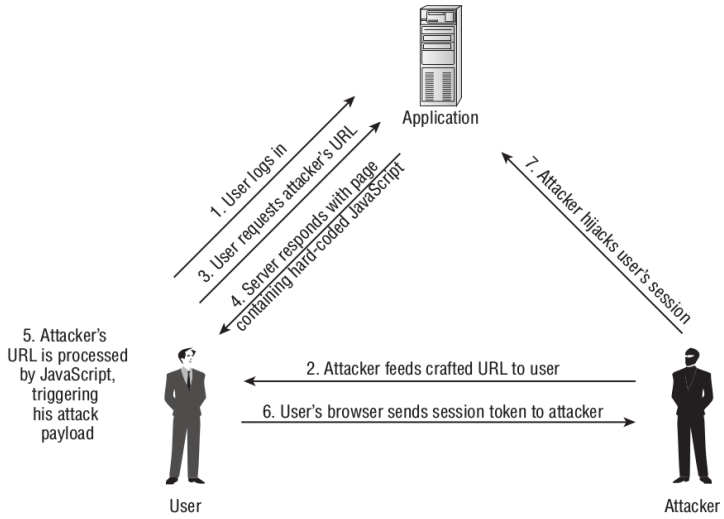

Figure 12-5: The steps involved in a DOM-based XSS attack

Figure 1.3: steps involved in DOM based XSS attack

small impact of advertisers potential for obtaining the data about users due to mechanisms such as super-cookies or fingerprinting.

There is also high social and disruption cost by certain types of XSS defacement attacks especially when combined with worm like behaviour.

We will note that the major reason why XSS is overlooked is that it doesn't offer the most precise way of targeting users or websites. often hackers prefer to opt for the more direct flaws which are ever-growing on the server or on the network to have more full control. Nevertheless in certain circumstances and if a hacker's purpose is not specific targeting XSS is a very reasonable approach that can yield great results and can allow a hacker to completely compromise an unprotected web application.

Typically mitigation is done through either reducing the risk of stealing the cookie (http-only , secure) or using some encrypted local store. As well as writting complicated Regular expression or evaluation based filters that sanitises inputs and parameters either through whitelist or blacklist ensures code does not leak into the origin and manage to run. The CSP policy can help drastically with this by restricting the script-src to disallow code to run from any remote origin, see CSRF 1.4.3.

1.3.2 CSRF

Unlike XSS which abuses your trust for a particular domain, Cross Site Request Forgery abuses your trust for the browser. What we mean by this is that CSRF relies on the fact that you are connected and have the cookie to a particular site that the hacker wants to attack. often for sites like facebook or google this is a reasonable assumptions since people leave them on when they go about their browsing. Then suffice it for the user to point to a domain where a hidden form has been injected or that is under the control of the hacker. he can perform unauthorised actions to the web application targeted abusing the fact that the browser will send the cookies as if authenticated. This is a serious threat, but relies on both having a good map of the functions of the target application and that the user is authenticated to the particular account you are targeting which is not of particular concern to sites where users are not often connected.

These kind of attacks came in these main flavours:

Session CSRF

The most common case is simply to somehow inject a hidden frame by tricking the user to your attack site (through deception or some kind of DNS poisoning) that performs a form submission using the token you had already authenticated hence causing you to perform an unwanted action Check 1.4 for illustration.

Login CSRF

This is an interesting variant, whose payload is a frame that authenticates the attacker's credentials to a service that the unknowing user uses, in that way exploiting the internal cohesion provided by Google for example, To get reports of the user's google search history on your interface. Check 1.5 for illustration.

Redirection CSRF attack

Where an attacker exploits flaws in certain designs where redirection is somehow parameterised and exploits tools like facebook connect which is used by many sites . Instead of redirecting the facebook token to the desired site for example yahoo.com it uses a yahoo internal server logic and queries yahoo.com/redirect?attacker.com to exploit the token from the attacker site.

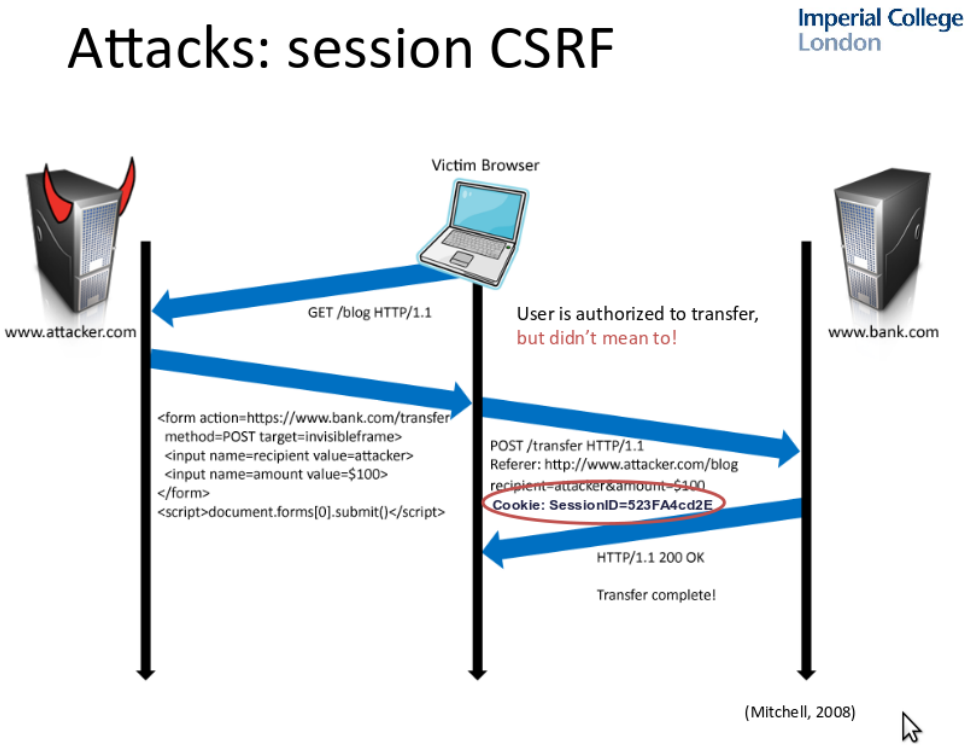

Figure 1.4: The typical session Cross Site Request Forgery

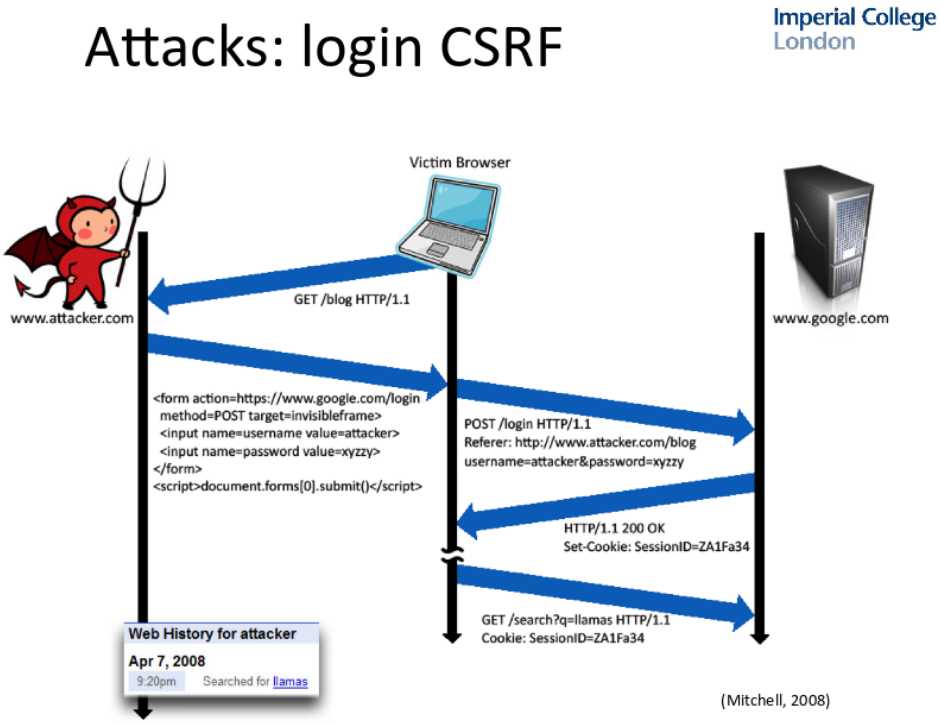

Figure 1.5: The login Cross Site Request Forgery lets attacker abuse interconnectivity of Google services

Social CSRF

This happens in the case of an affiliate service like CitySearch on facebook, The mechanism is that you are redirected from the attacker site to login to a service like facebook. And then since the affiliate could have a csrf vulnerability you abuse the vulnerability to propagate the csrf through facebook, in a sense exploiting not only the trust of the browser in facebook but the permissions granted by the trust between the affiliate and facebook. Check 1.6 for illustration.

More about CSRF

Remember that by using the GET methods where some of the above attacks, when it let's the user be redirected from the target site to your site allow to leak POST parameters through the refer header, the header that shows where you've been last , used mainly for advertisers and affiliate programs. This one of the reasons why POST is more secure in forms.

Note that there are also other possibilities of abuse, where for example a compromised machine has access to network services in the LAN, in which a CSRF would allow abuse of this privilege , but we do not go more in depth into this because our focus is the browser we will only cite the csrf soho router attack : [10].

These attacks are typically mitigated through use of csrf tokens in forms, which could be putting the very cookie as a form field or more securely a different token, usually the hash of the session and the desired action or something of the kind. Except in some edge cases like social csrf or redirection attack using the CSP policy connect-src directive would also block the attacker. from posting forms to sites that your site does not normally contact.

1.3.3 XST

We mentioned earlier in the cookie section that one way of mitigated the XSS attack is to use HTTP only cookie options. there are unfortunately a way around this with Cross Site Tracing, which uses the diagnostic HTTP trace method in a manner that allows it to retrieve the HTTP only cookie. If the cookie is secure as well (meaning encrypted) the chances become slim of obtaining the cookie.

1.3.4 Framebusting or clickjacking

These attacks are especially pervasive in the top websites. and consist in tricking the users into clicking on the content of another site which is placed in a visibility:hidden Iframe behind the current highlighted content. This is the reason for a Chrome and Firefox X-frame-Options header. see [6] for many real life example with twitter for example.

1.3.5 Other

Another overall privacy aware security policy in the browser is Refer Header (which is also source of XSS attacks potentially) and proposed HSTS mechanism we will evaluate effectiveness of those mechanisms and will see through that a compliant web browser which passes BrowserAudit could be made to minimze information leakage, of course the mechanisms that law enforcement (and criminals alike) might use on lower levels of the networking stack are outside the scope of this paper. We will also note that openSSL and WebCrypto implementation of the browser was not tested by BrowserAudit tool.

1.4 Security policies

1.4.1 Same Origin Policy

(SOP) governs the access control on today's browsers. The SOP prevents documents or scripts loaded from one origin from getting or setting properties of documents from a different origin . (The origin that a script is loaded is the origin of the document that contains the script rather than the origin that hosts the script.) Two pages have the same origin if the protocol, port (if given), and host are the same for both pages.

Each document is associated with an origin. The SOP policy concerns three browser resources: cookies, the HTML document tree, and remote store access. In more detail, a site can only sets its own cookie and a cookie is sent to only the site that sets the cookie along with HTTP requests to that site. Two documents from different origins cannot access each other's HTML document using the Document Object Model (DOM) which is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

1.4.2 Cross Origin Resource Sharing

As seen in te previous section User agents commonly apply same-origin restrictions to network requests. These restrictions prevent a client-side Web application running from one origin from obtaining data retrieved from another origin, and also limit unsafe HTTP requests that can be automatically launched toward destinations that differ from the running application's origin.

In user agents that follow this pattern, network requests typically include user credentials with cross-origin requests, including HTTP authentication and cookie information.

We mentioned in the history section that CORS extends SOP and it extends this model in several ways:

A response can include an Access-Control-Allow-Origin header, with the origin of where the request originated from as the value, to allow access to the resource's contents. The user agent validates that the value and origin of where the request originated match. User agents can discover via a preflight request whether a cross-origin resource is prepared to accept requests, using a non-simple method, from a given origin. This is again validated by the user agent.

Server-side applications are enabled to discover that an HTTP request was deemed a cross-origin request by the user agent, through the Origin header. This extension enables server-side applications to enforce limitations (e.g. returning nothing) on the cross-origin requests that they are willing to service. This specification is a building block for other specifications, so-called CORS API specifications, which define how this specification is used. Examples are Server-Sent Events and XMLHttpRequest. [EVENTSOURCE] [XHR]

If a resource author has a simple text resource residing at http://example.com/hello which contains the string "Hello World!" and would like http://hello-world.example to be able to access it, the response combined with a header introduced by this specification could look as follows:

Access-Control-Allow-Origin: http://hello-world.example

Hello World!

Using XMLHttpRequest a client-side Web application on http://hello-world.example can access this resource as follows:

```
var client = new XMLHttpRequest()
client.open("GET", "http://example.com/hello")
client.onreadystatechange = function() { /* do something */ }
client.send()
```

It gets slightly more complicated if the resource author wants to be able to handle cross-origin requests using methods other than simple methods. In that case the author needs to reply to a preflight request that uses the OPTIONS method and then needs to handle the actual request that uses the desired method (DELETE in this example) and give an appropriate response. The response to the preflight request could have the following headers specified:

```
Access-Control-Allow-Origin: http://hello-world.example\
Access-Control-Max-Age: 3628800\
Access-Control-Allow-Methods: PUT, DELETE\
```

The Access-Control-Max-Age header indicates how long the response can be cached, so that for subsequent requests, within the specified time, no preflight request has to be made. The Access-Control-Allow-Methods header indicates the methods that can be used in the actual request. The response to the actual request can simply contain this header:

```
Access-Control-Allow-Origin: http://hello-world.example
```

The complexity of invoking the additional preflight request is the task of the user agent. Using XMLHttpRequest again and assuming the application were hosted at http://calendar.example/app the author could use the following ECMAScript snippet:

```
function deleteItem(itemId, updateUI) {
  var client = new XMLHttpRequest()
  client.open("DELETE", "http://calendar.example/app")
  client.onload = updateUI
```

```
  client.onerror = updateUI
  client.onabort = updateUI
  client.send("id=" + itemId)
}
```

1.4.3 Content security Policy

The content security policy is the modern solution to the previous mess of SOP and CORS. It is a clean specification loaded with a page as a meta attribute or via an HTTP header. It allows to define custom made policies that specify which origins are allowed PER HTML entity/tag. meaning that one can say that scripts can only come from a particular set of domains and images from another. when used properly this hugely decreases the risk of CSRF, clickjacking or frambusting.

To mitigate XSS attacks, for example, a web application can declare that it only expects to load script from specific, trusted sources. This declaration allows the client to detect and block malicious scripts injected into the application by an attacker. check [3] for latest standard. DOM based XSS is still possible in some cases.

an example policy delivered in meta tag is

```
<meta http-equiv="Content-Security-Policy" content="script-src 'self'">
```

Multiple policies can be enforced which means that the most restrictive rules are applied example in the document containing both these policies.

```
Content-Security-Policy: default-src 'self' http://eg.com http://example.net;
                         connect-src 'none';
Content-Security-Policy: connect-src http://eg.com/;
                         script-src http://eg.com/
```

we cannot connect to http://eg.com because of the connect-src 'none'; rule. The Standard has report uri facility which allows to flag and report security violations to be sent to a defined URL.

```
Content-Security-Policy-Report-Only: script-src 'self';
                                     report-uri /csp-report-endpoint/
```

There is also another feature highlighted in this example which is the possibility of only reporting CSP violations. which could be a first line for iteratively defining the suitable policy for a web application that might not be aware of all the components he is using in modern web applications. Or

that might find offending behaviour that needs to be studied first before enforcing the policy.

1.4.4 X-frame-Options

Mentioned earlier this specifies which Origin is allowed to render the document in a frame. this can apply to ¡frame¿, ¡object¿, ¡applet¿ and ¡embed¿ elements. The options available for Origin are DENY , SAMEORIGIN, or ALLOW-FROM and this feature is supported in most modern browsers. It would certainly make a lot of streaming websites less annoying if they used this header but Perhaps also less lucrative,in the common case of ad clickjacking.

1.4.5 HSTS

HSTS defines a mechanism enabling web sites to declare themselves accessible only via secure connections and/or for users to be able to direct their user agent(s) to interact with given sites only over secure connections. The policy is declared by web sites via the Strict-Transport-Security HTTP response header field and/or by other means, such as user agent configuration, for example.

1.5 Adoption

We felt that the common plague of security in technology in general is relevant here as well, information about the security while widely available requires substantial effort for a novice or even seasoned developer to understand since the standards keep changing and browsers differ on priorities at the moment. There is also a need for more usability and for developers to know easily which server options to use to perhaps make it less frustrating and time consuming like setting up a new

There are also many subtle ways in which trust boundaries can be exploited and it's useful to think about how much more trust you have for libraries and mashups you allow in your site. and we cannot seperate. and often cases making a more lenient but robust security mechanism can mean the world. Because Hackers do not get frustrated with a certain restriction; and end up doing something silly from a security perspective like putting a foreign untested script within your trusted scripts or randomly adding

any domain to your CORS. We will need to acquire data about vulnerable browsers and adoption figures with things such as common policies, this is particularily hard to find and will be an objective of ours.

Header name	Description	Example	Status
Set-Cookie	An HTTP cookie Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1	Permanent: standard	
Status	CGI header field specifying the status of the HTTP response.	Status: 200 OK	Not listed as a registered field name
Strict-Transport-Security	A HSTS Policy informing the HTTP client how long to cache the HTTPS only policy and whether this applies to subdomains.	Strict-Transport-Security: max-age=16070400; includeSubDomains	Permanent: standard
X-Frame-Options[34]	Clickjacking protection: deny - no rendering within a frame, sameorigin - no rendering if origin mismatch, allow-from - allow from specified location, allowall - nonstandard, allow from any location[35]	X-Frame-Options: deny	Obsolete
X-XSS-Protection[39]	Cross-site scripting (XSS) filter	X-XSS-Protection: 1; mode=block	non-standard
Content-Security-Policy	X-Content-Security-Policy X-WebKit-CSP Content Security Policy definition	X-WebKit-CSP: default-src 'self'	Working draft

Table 1.2: List of response headers of interest to this paper

Attacks: social CSRF

Imperial College London

1. Alice signs in on CitySearch using Facebook
2. Her posts on CitySearch appear also on her Facebook newsfeed
3. CitySearch has a CSRF vulnerability on its review page
4. Facebook now inherits the CSRF, even if Alice won't visit CitySearch anymore

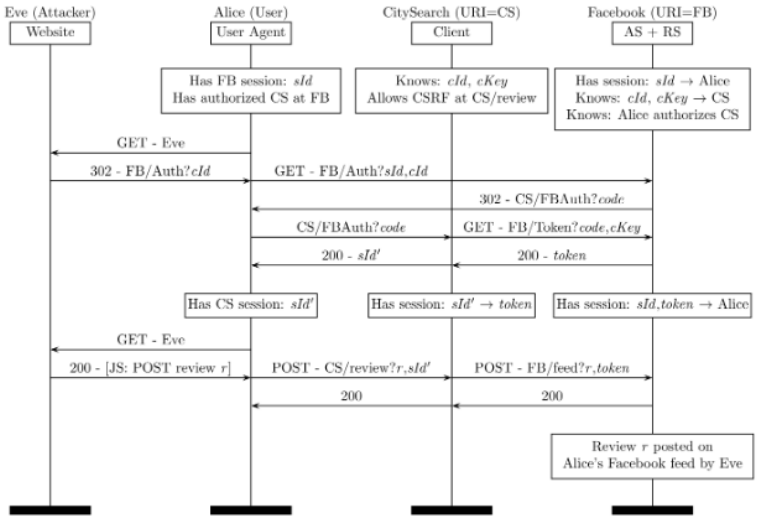

Figure 1.6: The Social csrf : discoverd through automated reasoning tools

Chapter 2

BrowserAudit as it stands

2.1 Initial project (V1)

Charlie Hothersall's thesis can be found in [11] , Browseraudit was built largely as a modification of Mocha with Chai assertion library Mocha was meant to be a nodejs testing library, but had browser support, hence allowed to have a base framework to save time and get the actual tests written

It was really thought of as a Unit Testing framework code running on the browser which was in a way a valid analogy to Unit tests that the browser developers would have written for the implementation of security policies for their browsers, but of course here done at the client level. While Mocha helped save some time for the task and it's asynchronous options were appreciated to save time running the tests; It did however need to be heavily customized with a rebuilt frontend using a Reporter interface that triggers callbacks on testStart end and Suite start and End.

The frontend was beautifully done, making use of twitter bootstrap, with suitable design that didn't get in the way of understanding the tests, and that offered expandable information about the tests from just the name and category and Pass/fail down to the exact code and the assertions that fail or errors matching different users capabilities.

There were a fair amount of basic tests of which the most notable one that illustrates an important tool for running these tests which is the img src property: you can see this in action in 2.1

As you can see the mechanism of img src property is used to communicate to the server on predefined appropriate pass/fail testiD endpoints. This is because the img object is not subject to any limitations and doesn't interfere in the testing of the security policy itself. This is used in many other types of tests.

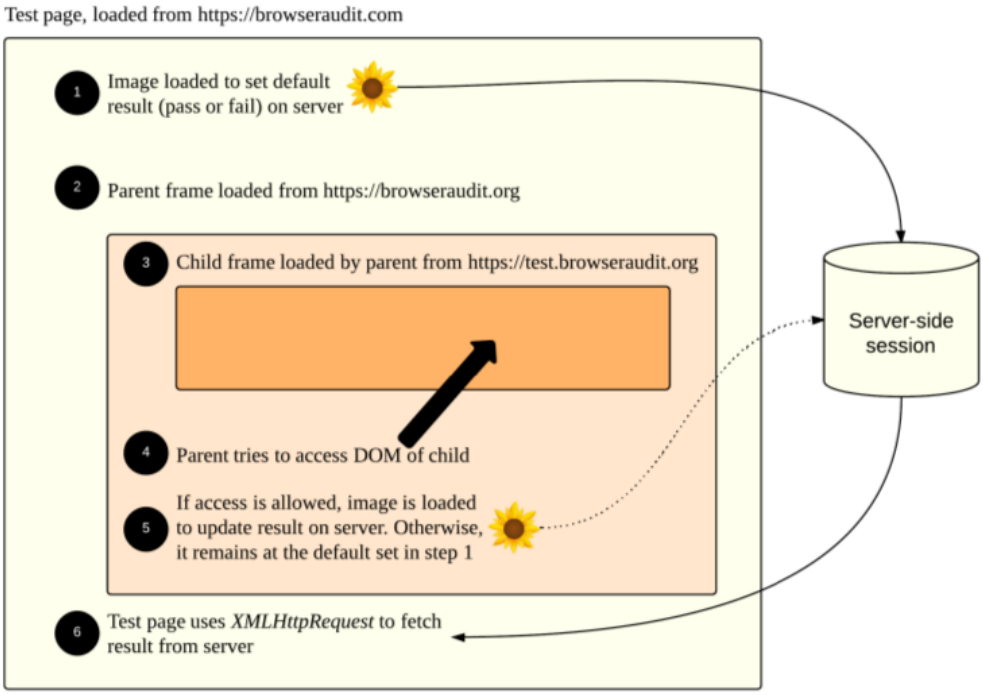

Figure 8: An example of a SOP test in which the parent frame tries to access the DOM of its child.

Figure 2.1: Same origin policy typical test

The initial version had over 300 test tests and by using plain javascript and dilligently avoiding methods that are not adopted by most browsers, The initial browseraudit version was supported and tested on these browsers:

- Firefox desktop version 26, 29: 4 /306 and 6/306 failures respectively
- Android version 30: 14 failures
- Ie10: 74/306 failures issues running the actual tests, the failures are test implementation issues.
- Ie11: 64/306 In this instance some of the tests that failed in ie10 run but they are still failures.
- Safari Desktop version 6 and Ios Version: 6 /286 runs fast with few errors, less tests because no HSTS.
- Android 4.2.2: 64/267 problems with CORS and AJAX implementation
- Android 4.4.2: 2/286 failing test due to X-frame-Options ALLOW FROM not implemented.

Most notable discoveries of browseraudit were firefox's CSP implementation: CSP allows local CSS @import with only 'unsafe-inline' set and CSP allows local Worker construction with only 'unsafe-inline' set a very good spot that would have been very hard to detect in the wild!

2.2 Current version adding tests (V2)

Dr Sergio maffeis and Chris Novakovic took the project in hand and improved coverage drastically, there are now up to 404 tests and counting. Matching the newer CSP standard, clickjacking and framebustig related tests etc..

The fact that Mocha was a nodejs package that had browser support started to show, because browser coverage is not terrific with the test framework. Everybody had forgotten ie6 but ie8+ is considered a modern browser by many and we aim to be able to test any configuration in the wild, and ie at the time of writing still represents 25+% of the marketshare.

For this reason the team decided to go for it's own test framework. which greatly simplifies the whole application, avoiding all the boiletplate that turned a unit testing framework into a browser based testing framework.

This current version , which we will refer to as (V2) is constantly being upgraded, and close integration with the code is required for my implementation but the new tiered architecture offers a simple modular way to deal with the tests which we will describe in the rest of this section.

There is an intermediate version, V1.5, if you will, which was still running mocha and chai for test running but had all the extra tests written and was in the draft version of the browseraudit paper , available at [7]. We will only focus on the latest version because the framework and tests are thoroughly described in [11]

2.2.1 Test Backend in V2

Overall execution model

Under the current version, categories, testsuites and tests are loaded from a postgres database, and test results "objects" are also saved on the database. This is a Backend centric architecture which writes js hooks to the frontend.. the execution object models are loaded onto go code which generate javascript on the fly. This javascript calls bridge functions similar to BrowserauditCategory(bla,bla,bla,bla) registering different categories and tests, it also passes their function invocation code with the correct html structures hardcoded as static assets.

This leaves to the frontend the sole mission to execute the code and update the UI with appropriate categories, descriptions, test suites, expected test outcomes and actual outcomes. Due to the way the tests are setup the outcome of those tests is known at the server endpoints as soon as the tests finish, leaving the go code that responds to the correct endpoint the task of also saving the feedback to the correct object in the database.

It is also worthwhile to note that the go server sets up appropriate endpoints for the tests with the right testid etc from the database reflecting the different policies being tested, with a general pattern of different modules handling different types of requests and hence adding certain headers or responding a particular payload etc.

PostgreSQL model

Take the time to check 2.2 the foreign keys have been conveniently marked with a diamond connector, we did not go into the details of suite execution settings. because it was containing identical data in terms of displaymode this is a yet unused provision for future test suite display options among other settings.

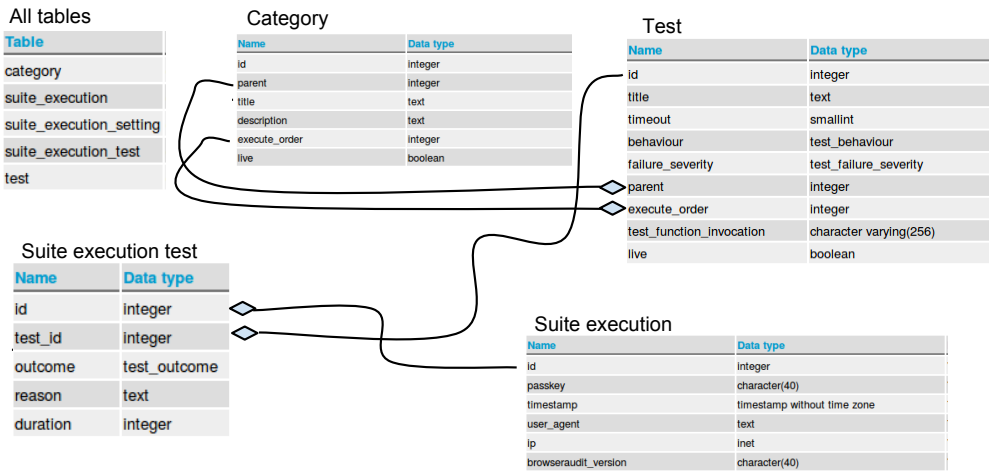

Figure 2.2: Database description of most important tables.

As you can see there is a clear distinction to make between tests which belong to catgories and executions of browserAudit on users machines and the outcome of tests during those executions. This follows what was explained earlier about the execution model. As you can see everything about the tests is stored in database, which is the most sensible approach for managing the sheer number of tests and providing maximum modularity concerning what we do with the data and how we add tests.

It seems the way tests are currently being added is manually to the database and this is manageable since the data model is simple enough, but the first step we will take is to improve on this and make our own interface to add tests in order to simplify all this, see 3.1.

The server

Nginx

There is an Nginx reverse proxy serving the static content and caching where cache directives are to allow caching for a response. this is thoroughly explained both in [11] and [7]. an Apache

the Go server

As we discussed earlier the Go server handles the responses at the appropriate endpoints for the types or categories of tests. As we said in the model section, what is actually saved in the database in terms of test execution is the invocation of the right function types with suitable parameters and

identification for later logging of results. We will now go into the details of the client code that handles those classes of tests in section 2.2.1

The Client test classes

The client test classes are shown in the following table with the literal function names and short description these are the methods called by the tests that are in the database:

- parentChildSopTest // Same-Origin Policy -¿ DOM access
- ajaxSopTest // Same-Origin Policy -¿ XMLHttpRequest
- domainScopeCookieTest // Same-Origin Policy -¿ Cookies - domain scope
- cookiePathScope // Same-Origin Policy -¿ Cookies - path scope
- cspTest //Content Security Policy
- originExpect // Cross-Origin Resource Sharing -¿ Access-Control-Allow-Origin
- methodExpect // Cross-Origin Resource Sharing -¿ Access-Control-Allow-Methods
- headersExpect// Cross-Origin Resource Sharing -¿ Access-Control-Allow-Headers
- exposeExpect // Cross-Origin Resource Sharing -¿ Access-Control-Expose-Headers
- cookiesHttpOnlyServerToScript // Cookies -¿ HttpOnly flag -¿ HTTP-only cookie set by server and accessed from JavaScript
- cookiesHttpOnlyScriptToServer // Cookies -¿ HttpOnly flag -¿ HTTP-only cookie set by JavaScript (should not be sent to server)
- cookiesSecureServerToScriptHTTPS // Cookies -¿ Secure flag -¿ cookie set by server should be sent over HTTPS
- cookiesSecureScriptToServerHTTP // Cookies -¿ Secure flag -¿ cookie set by JavaScript should not be sent over HTTP

- requestRefererHTTPSToHTTP // Request Headers -¿ Referer -¿ should not be sent over non-secure request if the referring page was transferred with a secure protocol
- frameOptionsTest // Response Headers -¿ X-Frame-Options
- hstsTest // Response Headers -¿ Strict-Transport-Security

Chapter 3

Design and Implementation

3.1 Improving extensibility and coverage

test Adding interface

adding new tests

we start simple, because simple is beautiful. The simplest types of tests we could thing of required a new category: Client-only tests. These tests do not require interaction from the server, except for reporting the results for a particular test execution.

we are talking mainly about javascript tests, postmessage on local frames, navigation in same window for example.

these tests are self-explanatory and allows us to guage the external quality of our test adding interface progressively, meaning first only needing to add client code to the database and registering the results of their execution on the client, bypassing the need to setup endpoints with particular types of responses or other complex configuration.

Strict mode javascript tests

The first test we're going to include deals with Strict Mode, it is a new feature in ECMAScript 5 that allows you to place a program, or a function, in a "strict" operating context. This strict context prevents certain actions from being taken and throws more exceptions. to use this feature suffice it to add the "use strict"; keyword at the top of the page , or very conveniently in our case , in the context of a function only. Using strict mode allows to raise errors on using certain bad practices of ES3 which ES5 is normally fully retroactively compatible with.

What is not allowed in strict mode, according to [12]:

- use of implicit or undefined globals will result in error.

- Deleting a variable, a function, or an argument will result in an error, instead of silently failing.
- defining a property more than once in an object literal will cause an exception to be thrown.
- Virtually any attempt to use the name 'eval' is prohibited – as is the ability to assign the eval function to a variable or a property of an object.
- Additionally, attempts to introduce new variables through an eval will be blocked.
- with() is a syntax error.

Since all of these are restrictions, we will trust that it doesn't over restrict and will just test for negative properties for each of these prohibited behaviour. And a

Hence, our first test is:

```
(function(){
 "use strict";
 try{
   bignametoavoidclash = 'initialising a global, -10000 marks!';
 }
 catch(bro){
   console.log("you shouldn't have done that" + bro);
 }
 }());
```

3.2 statistics page

3.3 Strech Goal: A DSL for test inputting and parameterisation

Chapter 4

Evaluation

From the planning section we identified that there are three main parts to the project with different fallbacks and here's how evaluation of those should proceed.

Our initial measure of success will be to empirically determine the effectiveness of different policy configurations on browsers to the latest security threats. We will do this against common browsers and the most recent browsers. This will be the way to quantitatively assess how much depth and breadth we managed to achieve in our underlying knowlege of the security considerations . Result can range from not finding many effective attacks to the latest browsers with latest browsers in which case we will still need to document what policies are effective for which cases, attempt to infer general patterns for use by other developers who might not afford the time and effort to do so. . On the other hand it is entirely possible to find massive flaws in the different policies or the implementation of new HTML5 features in browsers in which case we will have the added benefit of providing insight and advancing security for all. If things go this well, then it will be a good idea to start to build channels of communication with browser devs. or standards people to discuss the direction of standards which might give valuable insight.

As for the second part (building the data aggregator) this is a fairly mundane programming task which has well defined behaviour that we can test for with Functional tests The ease of use and design is the quality metric that will guage how creative I managed to get with it. and how many extra features were built in the time frame.

As for building some interactive infographic or something of the kind for making developers save time understanding and using these policies in practice this is more of an open ended goal, and can only measure success by the amount of excitement it generates and perhaps I could do with polling or

live testing in order to iteratively build this part. Another way to measure success is hit and bounce rate on the site itself for which we would use googleAnalytics or newrelic to obtain deep insight into what matters to web developers and security aware users.

Ultimately as a web developer it would be an added benefit and a first point of reference to build something that will make writing secure client based applications less of a hassle. And while precisely contains the cutting edge threats in one place gives also a manner of priorising effort compared to risks. Also giving examples for each types of intrustions in a digestable format would be an very beneficial.

Chapter 5

Conclusion

Bibliography

[1] *Same Origin Policy*, RFC 6454, available at http://tools.ietf.org/html/rfc6454

[2] *Uniform Resource Locator*, RFC 3986 , can check at http://www.ietf.org/rfc/rfc1738.txt

[3] *Content Security Policy Level 2* ,W3C Candidate Recommendation, 19 February 2015 can check at http://www.w3.org/TR/CSP/0

[4] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Webjail,*least-privilege integration of third-party components in web mashups. In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM*

[5] *Cross-Origin Resource Sharing*,W3C Recommendation 16 January 2014, check at http://www.w3.org/TR/2014/REC-cors-20140116/

[6] Gustav Rydstedt, Elie Bursztein, Dan Boneh, *Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites*

[7] Charlie Hothersall-Thomas, *BrowserAudit A web application that tests the security of browser implementations*, 2011

[8] Dr. Josh Pauli, *the web Application Hacker handbook ver.2*

[9] Michal Zalewski, *the Tangled web 2012.*

[10] *A Team Cymru EIS Report: Growing Exploitation of Small Office Routers Creating Serious Risks* , available at http://www.doc.ic.ac.uk/ maffeis/331/TeamCymruSOHOPharming.pdf

[11] *BrowserAudit A web application that tests the security of browser implementations*, available at

at http://www.doc.ic.ac.uk/teaching/distinguished-projects/2014/c.hothersall-thomas.pdf

[12] John Resig, *ECMAScript 5 Strict mode, JSON and More* , available at http://ejohn.org/blog/ecmascript-5-strict-mode-json-and-more/