KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

Bescherming van mobiele browsers tegen cross-site request forgery

Maarten Lambert

Thesis voorgedragen tot het
behalen van de graad van Master
in de ingenieurswetenschappen:
computerwetenschappen

Promotoren:
Dr. ir. L. Desmet
Prof. dr. ir. F. Piessens

Assessoren:
Prof. dr. D. Clarke
Dr. P. Philippaerts

Begeleiders:
P. De Ryck
Ir. S. Van Acker

Academiejaar 2010 – 2011

Voorwoord

Ik zou van deze gelegenheid willen gebruik maken om iedereen te bedanken die heeft bijgedragen tot de realisatie van deze masterproef.

In het bijzonder zou ik mijn promotor Lieven Desmet, co-promotor Frank Piessens en begeleiders Philip De Ryck en Steven Van Acker willen bedanken voor hun uitstekende begeleiding. Zonder hun gepaste bijsturing zou deze masterproef nooit de vorm en inhoud gekregen hebben die ze nu heeft.

Verder zou ik ook mijn familie en vooral mijn vriendin Veerle willen bedanken. Hun steun en vertrouwen gedurende het ganse jaar gaven me het extra steuntje in de rug dat nodig was om deze masterproef te verwezenlijken.

Maarten Lambert

Inhoudsopgave

Samenvatting

De laatste jaren maken steeds meer webtoepassingen gebruik van inter-domein communicatie om informatie te verkrijgen die andere domeinen ter beschikking stellen. De beveiligingsproblemen waarmee vele populaire webtoepassingen in het verleden geconfronteerd werden, geven echter aan dat de veiligheid van inter-domein communicatie geen sinecure is. Een belangrijke aanval die een grote impact kan veroorzaken is *Cross-Site Request Forgery* (CSRF). Bij deze techniek kan een aanvaller bepaalde verzoeken uitvoeren in naam van een andere gebruiker, zonder dat deze zich daar bewust van is. Verder is er een duidelijke groei wat betreft het aantal mobiele toestellen met toegang tot het internet. Deze apparaten laten eveneens inter-domein verkeer tussen website toe, waardoor ook hier beveiligingsmaatregelen onontbeerlijk zijn.

De bijdrage van deze masterproef is drievoudig. Ten eerste wordt onderzoek geleverd naar de veiligheid van inter-domein communicatie. Dit onderzoek bestaat enerzijds uit een kritische bespreking van bestaande CSRF maatregelen. Anderzijds bestaat ze uit een overzicht van twee recent ontwikkelde specificaties die veilig inter-domein verkeer willen verwezenlijken, *Cross Origin Resource Sharing* (CORS), reeds ondersteund door de meeste browsers, en *Uniform Messaging Policy* (UMP), een systeem dat momenteel nog geen browser ondersteuning geniet.

De tweede bijdrage bestaat uit de ontwikkeling van een CSRF beschermingsmaatregel voor browsers van mobiele platformen in de vorm van een extensie aan de client-zijde. Deze vertrekt vanuit de werking van CsFire, een bestaande beveiliging voor Mozilla Firefox. Enerzijds wordt in deze masterproef een extensie voor Mozilla Fennec ontwikkeld, de mobiele versie van de populaire browser Mozilla Firefox. Dit is de eerste extensie voor Mozilla Fennec die rekening houdt met bestaande web 2.0 technieken en het gebruik hiervan zo minimaal mogelijk tracht te verstoren. Anderzijds brengt het onderzoek in deze masterproef ook een extensie voor de Android Browser voort, die standaard geïnstalleerd staat op alle Android toestellen. Dit is een primeur aangezien er nog geen CSRF maatregelen bekend zijn voor deze browser.

Een laatste bijdrage bestaat uit de combinatie van de eerste twee bijdragen. De bestaande extensies blokkeren de werking van *Cross Origin Resource Sharing* geheel of gedeeltelijk waardoor websites deze techniek niet meer kunnen toepassen. De extensies die uit de tweede bijdrage voortvloeiden, worden dan ook aangepast zodat ze een gepaste ondersteuning bieden voor CORS. De aanpassingen houden in dat de bestaande CORS werking lichtjes gewijzigd wordt, waardoor een veiligere variant op dit mechanisme ontstaat.

Lijst van figuren

Lijst van tabellen

Lijst van afkortingen

Afkortingen

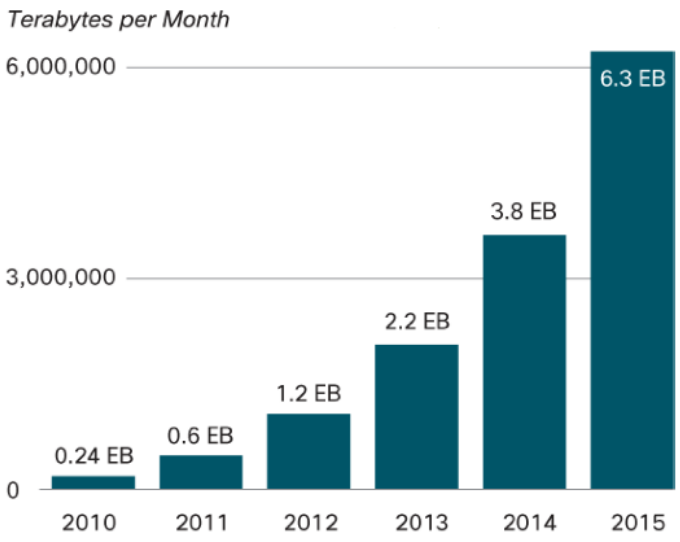
CSRF	Cross-Site Request Forgery
CORS	Cross-Origin Resource Sharing
UMP	Uniform Messaging Policy
EV	Eenvoudig Verzoek
AV	Geavanceerd Verzoek
ACAO	Access-Control-Allow-Origin
ACAM	Access-Control-Allow-Methods
ACAH	Access-Control-Allow-Headers
ACAC	Access-Control-Allow-Credentials
ACMA	Access-Control-Max-Age

Hoofdstuk 1

Inleiding

1.1 Opkomst van mobiele apparaten

De populariteit van mobiele apparaten met toegang tot het internet, neemt ziende ogen toe. Zo kent de zogenaamde *smartphone* al enkele jaren een stevige opmars en wint ook de *tablet pc* steeds meer aandacht van consumenten. De resultaten van een onderzoek uitgevoerd door Gartner [32], bevestigen deze waarnemingen. Het onderzoek schat het aantal desktop computers dat in 2013 in gebruik zal zijn op ongeveer 1.78 miljard. Dit aantal zal in dat jaar voorbijgestoken worden door het aantal mobiele apparaten, dat geschat wordt op 1.82 miljard.



FIGUUR 1.1: Een voorspelling van het wereldwijd mobiel dataverkeer van 2010 tot 2015 [15].

Naast een stijging van het aantal mobiele apparaten, is er ook een toename van het gemiddeld internetverkeer per mobiele gebruiker. Zo is volgens Cisco het

	GET	POST	Ander	Totaal
Inter-domein verkeer (strikte origine)	320529 (42,72%)	1793 (0,24%)	0 (0,00%)	322322 (42,96%)
Inter-domein verkeer (afgezwakte origine)	242084 (32,26%)	1503 (0,20%)	0 (0,00%)	243587 (32,46%)
Alle verkeer	722298 (96,26%)	28025 (3,74%)	11 (0,00%)	750334 (100,00%)

Tabel 1.1: Resultaten van een onderzoek naar inter-domein internetverkeer [48].

gemiddeld dataverkeer per mobiel toestel in 2010 verdubbeld tot bijna 80 megabyte per maand [15]. Datzelfde onderzoek voorspelt dat het globaal mobiel verbruik tegen 2015 ongeveer 6,3 exabyte[1] zal bedragen, wat neerkomt op een stijging met een factor 26 ten opzichte van het verbruik in 2010. Figuur 1.1 geeft de evolutie van 2010 tot 2015 weer, zoals voorspeld door Cisco.

De toename van mobiel dataverkeer wijst erop dat een onderzoek naar mobiele platformen, met name naar de veiligheid van deze platformen, niet mag uitblijven. Deze masterproef legt zich dan ook toe op het onderzoek naar dit type apparaten.

1.2 Nood aan veilige inter-domein communicatie

Inter-domein communicatie, of *cross-domain communication*, verwijst naar de communicatie die binnen de browser plaatsvindt tussen websites die zich op verschillende internet domeinen bevinden. Een verzender kan een verzoek, ook *request* genoemd, verzenden naar een bepaalde ontvanger. De ontvanger wordt geïdentificeerd door een afkomst of origine. Of een bepaald verzoek al dan niet inter-domein gebeurt, hangt af van de definitie van een origine. Vaak wordt een origine bepaald door een domeinnaam, een protocol en een tcp poort. In `http://example.com:80` wordt *example.com* als de domeinnaam, *http* als het protocol en *80* als de tcp poort beschouwd. Indien één van deze elementen verschilt, is er sprake van een inter-domein verzoek.

Mede door de opkomst van online toepassingen is het aandeel inter-domein verzoeken ondertussen aanzienlijk toegenomen. Een analyse van internetverkeer bij 15 proefpersonen gedurende 2 weken, leidde tot de resultaten in tabel 1.1 [48]. Het onderzoek maakte onderscheid tussen enerzijds de strikte definitie van origine (domeinnaam, protocol en poort) en een afgezwakte definitie (domeinnaam). Indien de eerste definitie gebruikt wordt, blijkt 43% van het verkeer inter-domein te zijn, de afgezwakte definitie geeft 32% aan. Inter-domein communicatie is dus een veelvoorkomend fenomeen en heel wat websites zijn afhankelijk geworden van gegevens die zich op andere domeinen bevinden.

[1] 1 exabyte is gelijk aan 1 miljoen terrabytes

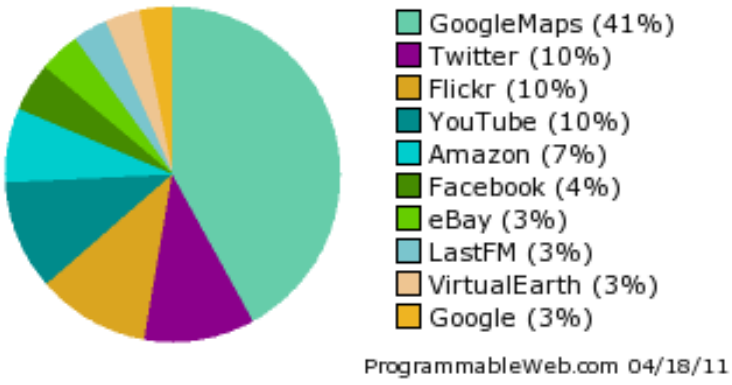
Deze stijging van inter-domein verkeer valt deels te verklaren door de opkomst van zogenaamde *web mashups*. Een mashup is een webapplicatie die verschillende diensten van andere domeinen aanspreekt. Deze gecombineerde informatie leidt tot een meerwaarde voor de gebruiker van de applicatie. Steeds meer online diensten stellen hun informatie open tot het publiek via een API. Tijdens dit schrijven zijn er minstens 3206 API's actief die door mashups gebruikt kunnen worden en minstens 5790 mashup applicaties die van deze diensten gebruik maken[2]. Figuur 1.2 toont een overzicht van de meest gebruikte API's.

Enkele voorbeelden van mashups:

- *iGoogle:* Een bekend voorbeeld van een mashup is de service http://www.igoogle.com die dienst kan doen als een gepersonaliseerd webportaal. Het bevat mogelijkheden om informatie van andere domeinen te verkrijgen en samen te brengen in één pagina. Een startpagina kan op deze manier bijvoorbeeld weerinformatie, nieuwsinformatie en wisselkoersinformatie samenbrengen.

- *Parking beschikbaarheid:* De website http://www.parkingcarma.com geeft de aanwezigheid van vrije parkeerplaatsen aan in verschillende steden in de Verenigde Staten. Ze maakt hiervoor gebruik van *Bing Maps* om de informatie visueel voor te stellen.

- *Immobiliën:* Veel immo websites combineren online beschikbare landkaarten met de adressen van panden die te koop of te huur zijn. Op die manier kan de gebruiker eenvoudig buurten doorzoeken naar mogelijke huizen of appartementen. Een voorbeeld hiervan is terug te vinden op de website http://www.housingmaps.com.



FIGUUR 1.2: De top 10 API's gebruikt door mashups volgens http://www.programmableweb.com.

[2]Deze gegevens werden verkregen op 10 mei 2011 van de website http://www.programmableweb.com

1.3 Doel van deze masterproef

Deze masterproef heeft als eerste doel na te gaan in hoeverre de huidige browsers veilig inter-domein verkeer toelaten en welke mechanismen er bestaan om dit verkeer op een veilige manier toe te laten. Hoofdstuk 2 behandelt een belangrijk veiligheidsprobleem dat betrekking heeft op inter-domein communicatie, *Cross-Site Request Forgery* (CSRF). Het gaat in op de werking van CSRF en op bestaande beveiligingsmechanismen. Vervolgens geeft hoofdstuk 3 een overzicht van enkele recent ontwikkelde technieken die als doel hebben veilig inter-domein verkeer toe te laten.

Het tweede doel van deze masterproef bestaat uit de ontwikkeling van een CSRF maatregel aan de client-zijde, die rekening houdt met de inter-domein communicatie mechanismen uit hoofdstuk 3. De bijdrage bestaat enerzijds uit een implementatie van een extensie voor Mozilla Fennec, de mobiele variant van Mozilla Firefox[3]. Anderzijds wordt ook een beveiligingsmaatregel ontwikkeld voor de Android Browser, de browser die standaard aanwezig is op het Android platform[4].

Hoofdstuk 4 tekent een algemene aanpak en architectuur uit, na een samenvatting van de probleemstelling die voortvloeit uit het onderzoek in hoofdstuk 2 en 3. Hoofdstuk 5 past deze aanpak toe op zowel het Mozilla platform als het Android platform.

Hoofdstuk 6 evalueert de ontwikkelde beveiligingsmaatregelen en vergelijkt de resultaten met de bestaande systemen die hoofdstuk 2 besprak.

Tot slot vat hoofdstuk 7 de belangrijkste resultaten van deze masterproef samen en zet ze de verschillende elementen van deze masterproef, die tot een verbeterde beveiliging van mobiele apparaten bijdragen, nog eens op een rij.

[3] Meer informatie over Mozilla Fennec kan teruggevonden worden op http://www.mozilla.com/en-US/mobile/

[4] Meer informatie over de Android Browser en het Android platform kan teruggevonden worden op http://www.android.com/

Hoofdstuk 2

Cross-Site Request Forgery

Dit hoofdstuk bespreekt *Cross-Site Request Forgery* (CSRF), een aanvalstechniek die mogelijk gemaakt wordt door een combinatie van een gebrek aan doordachtheid bij het design van de huidige browsermechanismen en een onvoldoende beveiliging van webapplicaties. Sectie 2.1 begint met een inleiding tot enkele bestaande browsermechanismen die betrekking hebben op inter-domein communicatie. Vervolgens bespreekt sectie 2.2 hoe hierdoor een CSRF aanval mogelijk gemaakt wordt en geeft ze aan hoe verschillende populaire websites het slachtoffer werden van deze aanvalstechniek. Tot slot bespreekt sectie 2.3 enkele bestaande beveiligingstechnieken en -maatregelen, die als verweermiddel kunnen dienen voor een CSRF aanval.

2.1 Bestaande browsermechanismen

2.1.1 Same Origin Policy

Een browser kan websites inladen van verschillende domeinen. Om de privacy van de gebruiker niet te schenden en bepaalde vertrouwelijke elementen van een domein af te schermen van andere domeinen, werd het *Same Origin Policy* (SOP) geïntroduceerd.

Over het algemeen laat het SOP niet toe informatie in te lezen of weg te schrijven naar een ander domein. Stel dat volgende twee websites zijn ingeladen in de browser: `http://foo.example.com` en `http://bar.example.com`. Dan kan de eerstgenoemde niets inlezen of veranderen aan de gegevens die voor de tweede website werden ingeladen en omgekeerd.

Wanneer `http://foo.example.com` bijvoorbeeld een afbeelding inlaadt van `http://bar.example.com` via volgende code, wordt deze wel getoond aan de gebruiker, maar kan de eerstgenoemde website geen pixels of andere afbeeldinginformatie inlezen of aanpassen.

```
<img src="http://bar.example.com/image.jpg" alt="an example image" />
```

Het SOP mechanisme laat echter een uitzondering toe op deze regel. Wanneer onderstaande code een script inlaadt vanuit een ander domein, neemt de website het script integraal op in zijn eigen domein. Het script maakt op dat moment niet langer

deel uit van het domein waarvan het gehaald werd. Deze eigenschap van het SOP heeft in het verleden tot mechanismen geleid die een bepaalde vorm van inter-domein communicatie mogelijk maken. Hoofdstuk 3 gaat verder in op de werking van deze mechanismen.

```
<script src="http://bar.example.com/script.js" alt="an example script" />
```

Het SOP beperkt dus de mogelijkheden om gegevens inter-domein in te lezen en aan te passen. Het verbiedt echter niet het maken van verzoeken zoals vorige voorbeelden aangaven. Sectie 2.2 zal aantonen dat dit belangrijke gevolgen heeft voor de veiligheid. De meeste grote browsers, waaronder Mozilla Firefox, Internet Explorer, Google Chrome en Safari, ondersteunen een bepaalde variant van het *Same Origin Policy*.

2.1.2 Authenticatie mechanismen

Het gebruikte protocol bij het opvragen van webpagina's in de browser wordt het *HyperText Transfer Protocol* (HTTP) genoemd. HTTP is een staatloos protocol wat inhoudt dat geauthenticeerde sessies bovenop dit protocol gebouwd moeten worden [19]. Websites maken hiervoor vaak gebruik van *cookies* of van de *HTTP authenticatie* methode [57, 62].

Cookies zijn bepaalde variabelen met een bijhorende waarde die een website kan vastleggen via de `Set-Cookie` hoofding. Volgend voorbeeld stelt een hoofding voor van een respons dat een uniek nummer toekent aan een gebruikerssessie:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: session-id=8765FJKHBU5YUIOIUY45;
```

De browser slaagt deze cookie op en voegt de informatie weer toe aan elke nieuw verzoek naar het overeenkomstige domein. In dit geval zou een volgend verzoek er als volgt kunnen uitzien:

```
GET /page.html HTTP/1.1
Host: example.org
Cookie: session-id=8765FJKHBU5YUIOIUY45;
```

De browser bewaart dus bepaalde gegevens die enkel gelezen en aangepast mogen worden door webpagina's van dezelfde origine. Websites kunnen deze sessie informatie gebruiken voor allerlei doeleinden zoals de identificatie van een gebruiker. De browser verwijdert de cookies eens de sessie is afgelopen, wanneer een gebruiker zich bijvoorbeeld afmeldt bij een website.

Een gelijkaardig systeem maakt gebruik van de *HTTP authenticatie* methode. In plaats van de cookie hoofding, maakt de browser in dit geval gebruik van een `Authorization` hoofding. Omwille van de sterke gelijkenis gaat de rest van de tekst

hier niet meer op in. De documentatie biedt meer informatie indien dit gewenst zou zijn [57].

Het SOP verbiedt het inlezen van cookies en HTTP authenticatie informatie door websites die niet tot hetzelfde domein behoren. Zoals de vorige sectie aangaf, kan een website echter wel een inter-domein verzoek genereren. Indien er sessie informatie aanwezig is, verzendt de browser deze impliciet mee met het verzoek. Dit geeft echter aanleiding tot *Cross-Site Request Forgery*, het beveiligingsprobleem dat de volgende sectie bespreekt.

2.2 De werking van CSRF

Cross-Site Request Forgery (CSRF) is een aanval die de integriteit van een sessie tussen een gebruiker en een vertrouwde website verstoort [14]. Deze aanval is gekend onder heel wat synoniemen zoals *Session riding*, *One click attack*, *Sea surf* en *Confused deputy* [34].

Het gevaar van CSRF volgt rechtstreeks uit de bespreking van de browsermechanismen in de vorige sectie. Uit sectie 2.1.1 bleek namelijk dat het *Same Origin Policy* niet vermijdt dat websites verzoeken kunnen verzenden naar eender welk domein. Zo kan een website met slechte bedoelingen bepaalde acties veroorzaken op een willekeurige website zonder medeweten van de gebruiker. De aangesproken server heeft op deze manier geen mogelijkheid om de oorsprong van deze verzoeken te controleren. Zowel gebruikers als andere websites kunnen namelijk een inter-domein verzoek initiëren.

De impact van CSRF vergroot bij het in rekening brengen van de bevindingen uit sectie 2.1.2 omtrent authenticatie mechanismen. Daaruit bleek dat bij het bestaan van een sessie via *cookies* of de *Http Authenticatie* methode, de browser deze informatie impliciet mee verzendt bij elk inter-domein verzoek. Het gebruik van deze vertrouwelijke identificatie mechanismen is dus niet beschermd tegen CSRF aanvallen. Bij de bespreking van CSRF aanvallen veronderstelt deze tekst het gebruik van een dergelijk authenticatie systeem.

2.2.1 Aanvalsvectoren

Nadat een gebruiker is ingelogd met zijn gebruikersgegevens, heeft een aanvaller heel wat mogelijkheden om een CSRF aanval succesvol uit te voeren. De simpliciteit van de aanval hangt af van het type verzoek en hoe gemakkelijk de gebruiker te misleiden valt om een kwaadaardige pagina te openen. [67].

- **Het type verzoek**
  - *Verzoeken die gebruik maken van de GET methode*

    Een verzoek dat gebruik maakt van de `GET` methode zou volgens de W3C specificatie geen veranderende effecten mogen veroorzaken [19]. In realiteit blijken vele ontwikkelaars daar echter geen rekening mee te houden [48]. Bij het gebruik van de `GET` methode moeten eventuele parameters

achteraan de Uniform Resource Identifier (URI) toegevoegd worden. Een `GET` verzoek kan zowel afkomstig zijn van HTML code als JavaScript code waardoor simpelweg Javascript uitschakelen geen voldoende bescherming biedt.

Een voorbeeld van een CSRF aanval via de `GET` methode:

```
<img src="foo.example.com/action=doSomeThing" style="display:none;" />
```

Dit voorbeeld maakt de ingeladen afbeelding bovendien onzichtbaar voor gebruikers wat de aanval transparant kan laten verlopen.

– *Verzoeken die gebruik maken van de POST methode*

Een `POST` verzoek kan verzonden worden via het `form` element. Opdat een CSRF aanval kan plaatsvinden, moet de aanvaller het formulier via een JavaScript automatisch laten verzenden.

Volgend voorbeeld, dat zich op het domein `bar.example.com` bevindt, bevat een JavaScript dat een dergelijke aanval uitvoert:

```
<form name="myform" action="foo.example.com" method="POST" >
      <input type="hidden" name="action" value="doSomeThing" />
      <input type="submit" name="finish" value="doFinishing" />
</form>

<script>document.myform.submit();</script>
```

De laatste lijn uit het voorbeeld zorgt voor de automatische verzending van het formulier zonder dat gebruikersinteractie vereist is.

– *Overige methodes*

Tabel 1.1 gaf reeds aan dat verzoeken bijna uitsluitend de `GET` of `POST` methode gebruiken. Het XMLHTTPRequest (XHR) object laat echter ook het gebruik van volgende methodes toe [61]: `CONNECT`, `DELETE`, `GET`, `HEAD`, `OPTIONS`, `POST`, `PUT`, `TRACE` en `TRACK`. Het onderstaand voorbeeld maakt gebruik van de `PUT` methode om een file te verzenden naar de server. De XHR API is beperkt tot het verzenden van verzoeken binnen hetzelfde domein wat CSRF aanvallen onmogelijk maakt. Het gaat hier wel om de XMLHttpRequest Level 1 API terwijl veel browsers recent de level 2 versie van deze API hebben aangenomen. Uit hoofdstuk 3 zal blijken dat deze versie wél inter-domein verkeer ondersteunt, maar op een veiligere manier dan de andere DOM elementen.

```
<script>
    var invocation = new XMLHttpRequest();
    invocation.open('PUT', url, true);
    invocation.onreadystatechange = handler;
    invocation.send(file);
</script>
```

- **De gebruiker misleiden**

  Na de ontdekking van een CSRF kwetsbaarheid, moet de aanvaller een manier vinden om de gebruiker te misleiden. De gebruiker moet een webpagina openen die het kwaadaardige inter-domein verzoek verzendt. Dit zal enerzijds afhangen van de voorzichtigheid waarmee de gebruiker e-mails en websites opent, anderzijds van de eventuele kwetsbaarheden van de websites die hij bezoekt. Volgende technieken zijn enkele van de vele manieren waarop een aanvaller dit deel van de CSRF aanval kan uitvoeren.

  – *Social engineering*

    De eerste methode bestaat erin de gebruiker te overtuigen op een bepaalde link te klikken door hem een e-mail of *chat-bericht* te verzenden. Deze vorm van misleiding komt meestal voor onder de noemer *social engineering*. De aanvaller stelt de boodschap op zo een manier op, dat de gebruiker zelf op de kwaadaardige link klikt. Om de misleiding een hogere kans tot slagen te geven kan hij de URL eventueel vervormen of verkorten. Voorzichtige gebruikers zullen in dit geval beter beschermd zijn, aangezien ze waarschijnlijk minder snel vreemde mails of links vertrouwen [42].

  – *Kwaadaardige websites*

    Een website waarvan de gebruiker zelf kwaadaardige bedoelingen heeft, kan gemakkelijk misbruikt worden om allerlei verborgen inter-domein verzoeken te sturen. Aangezien de website onder volledige controle staat van de aanvaller, is elk type CSRF aanval mogelijk. Het zal echter afhangen van andere technieken (bv. social engineering) om het slachtoffer te overtuigen zijn website te openen.

  – *XSS injecties*

    Een derde methode verloopt transparant zonder medeweten van het slachtoffer en de eigenaar van de website die het eigenlijke inter-domein verzoek verstuurt. Deze laatste bevat een zogenaamde XSS kwetsbaarheid die een aanvaller kan uitbuiten [43]. XSS, wat staat voor *Cross-Site Scripting*, zorgt ervoor dat een willekeurig script geïnjecteerd kan worden in de website. Wanneer een gebruiker deze vertrouwde website opent, voert de browser het script uit in de browser. Dit kan heel wat gevolgen hebben waaronder ook de uitvoering van een CSRF aanval naar een andere vertrouwde website. Een volledige beveiliging tegen XSS kwetsbaarheden is dus vereist maar dit is niet altijd even eenvoudig te bereiken. Er bestaan namelijk meer dan 100 aanvalsvectoren voor de uitbuiting van XSS [24]. De volgende sectie geeft een praktisch voorbeeld van een CSRF aanval die gebruik maakt van een XSS kwetsbaarheid.

2.2.2 Voorbeeld scenario

Deze sectie verduidelijkt de werking van een CSRF aanval via een voorbeeldscenario. Figuur 2.1 geeft een schematisch overzicht weer.

Figuur 2.1: Een schematisch overzicht van een typische CSRF aanval.

- **Stap 1: Bob logt in op de website van zijn bank**

  In deze eerst stap meldt de gebruiker Bob zich aan bij de bank. Het daarvoor bestemd formulier maakt gebruik van de `GET` methode en bevat twee invulvelden voor de gebruikersnaam en het paswoord van Bob. Hij vult deze gegevens in en klikt op de `invoer` knop. De browser verzendt vervolgens een verzoek naar de website met de bijhorende login informatie.

- **Stap 2: De bank beantwoordt het verzoek**

  De bank controleert de ingevoerde gegevens van Bob en besluit dat Bob het correcte paswoord heeft ingevoerd dat bij zijn gebruikersnaam hoort. Het antwoord van de server bevat een unieke cookie waarde die als identificatie van de gebruiker dienst doet. De browser slaagt deze waarde op en voegt ze toe aan elk nieuwe verzoek naar de bank. Op deze manier hoeft Bob zich niet elke keer opnieuw in te loggen.

  De website van de bank biedt Bob vervolgens de mogelijkheid om zijn rekeninginformatie te bekijken en eventuele geldoverdrachten uit te voeren. Hij heeft volledig vertrouwen in de veiligheid van de bank en beseft niet dat een voldoende CSRF beveiliging ontbreekt.

- **Stap 3: Bob bezoekt een forum met XSS kwetsbaarheden**

  Een tijdje later bezoekt Bob zijn favoriete forum terwijl de sessie met de bank nog steeds actief is. Bob is opnieuw niet op de hoogte van de beveiligingspro-

blemen met deze website en beseft niet dat er geen voldoende bescherming is tegen XSS aanvallen. Een malafide gebruiker kan zonder veel problemen een zelfgeschreven script injecteren in de website.

In dit geval heeft de aanvaller Mallory het volgende script in haar post geplaatst:

```
<iframe src="bank.com?action=transfer&target=mallory&amount=5000"
style="display;none" />
```

- **Stap 4: Het forum beantwoordt het verzoek voor het lezen van Mallory's post**

  Bob bekijkt enkele posts op het forum waarbij verschillende verzoeken en antwoorden gegenereerd worden tussen de browser en het forum. Op een bepaald moment wil Bob de post van Mallory bekijken. Het antwoord naar Bob's browser bevat naast de gewoonlijke informatie ook het geïnjecteerde script met kwade bedoelingen.

- **Stap 5: CSRF exploitatie**

  Het hoger beschreven `iframe` object voert onopgemerkt een inter-domein verzoek uit naar de website van de bank via de `GET` methode. Aangezien de sessie tussen de bank en Bob's browser nog steeds actief is, voegt de browser de cookie waarde toe aan het verzoek. De bank ontvangt het verzoek en veronderstelt dat deze het gevolg is van Bob's activiteit met de website. Dit is echter niet het geval en de bank schrijft een bedrag van 5000 euro over naar de rekening van Mallory.

  De CSRF aanval is bij deze succesvol ten einde gebracht zonder medeweten van Bob. Hij zal later versteld staan wanneer hij zijn rekeninginformatie bekijkt, aangezien hij niet weet hoe deze overdracht heeft kunnen plaatsvinden.

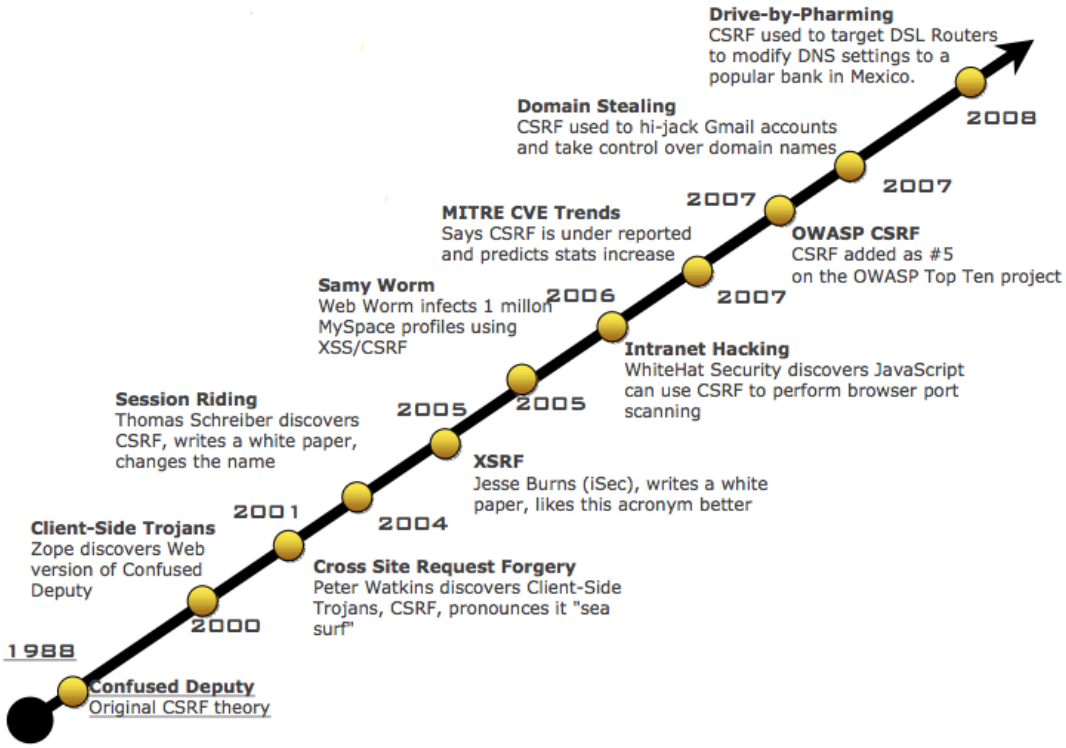
2.2.3 Historisch overzicht

Ondanks de mogelijk zware gevolgen van *Cross-Site Request Forgery*, heeft deze aanvalstechniek nog maar recent de nodige aandacht verworven. Deze sectie beschrijft de geschiedenis van CSRF, weergegeven in figuur 2.2.

De eerste publicatie met de term CSRF verscheen in 2011 in een artikel van Peter Watkins [25], die op zijn beurt verwees naar het werk van Norman Hardy, betreffende de *Confused Deputy theory* [26]. Watkins's publicatie bevatte reeds een gedetailleerd inzicht in CSRF kwetsbaarheden. Zo haalde hij enkele kritische bedenkingen aan bij mogelijke beschermingsmaatregelen, zoals het gebruik van merktekens, of *tokens*[1].

In 2004 vervolgde Thomas Schreiber het onderzoek naar CSRF. Hij merkte in zijn werk op dat heel wat bestaande webapplicaties kwetsbaar waren voor CSRF aanvallen [50]. Blijkbaar negeerden heel wat ontwikkelaars de dreiging van CSRF, ondanks het groeiende risico.

[1] Sectie 2.3.1 gaat nader in op het gebruik van merktekens als beveiligingstechniek.



Figuur 2.2: Een historisch overzicht van CSRF, gepresenteerd op de RSA conferentie in 2008 door Jeremy Grossman [23]

In 2006 verwees Jeremy Grossman naar CSRF als de *Sleeping Giant*, oftewel de slapende reus [22]. *Reus* slaagt op de grote impact die een CSRF aanval kan hebben. De volgende sectie somt een lijst van aanvallen die de afgelopen jaren plaatsvonden en waarin de relevantie van het begrip *Reus* bevestigd wordt. Toen Grossman zijn werk schreef, bestonden er heel wat onbeveiligde applicaties die tot op dat moment nog geen slachtoffer waren van CSRF aanvallen. Daarom voegde hij ook nog de term *Slapende* toe aan de beschrijving van een CSRF kwetsbaarheid.

In 2007 legden de beveiligingsorganisaties CVE en OWASP een verhoogde nadruk op CSRF door respectievelijk een stijging van CSRF aanvallen te voorspellen en het op de vijfde plaats in de OWASP top tien te plaatsen [23, 40]. Tabel 2.1 geeft aan dat CSRF ook in 2010 nog steeds de vijfde plaats van de OWASP top 10 inneemt en daarmee dus een belangrijke bedreiging blijft voor hedendaagse webapplicaties [44].

2.2.4 Recente aanvallen

Ondanks het groeiende bewustzijn voor CSRF kwetsbaarheden, slaagden veel bedrijven er de afgelopen jaren niet in hun websites voldoende te beveiligen tegen CSRF aanvallen. Dit blijkt uit volgende recente aanvallen op populaire websites [16, 67]:

Plaats	Kwetsbaarheid
1	Injecties
2	Cross-Site Scripting (XSS)
3	Gebroken authenticatie en sessie management
4	Onveilige directe object referenties
5	Cross-Site Request Forgery (CSRF)
6	Foutieve configuratie van de beveiliging
7	Onveilige cryptografische opslag
8	Falen de URL toegang te beperken
9	Onvoldoende beveiliging van de transportlaag
10	Ongevalideerde omleidingen naar andere webpagina's

TABEL 2.1: Top 10 Kwetsbaarheden volgens OWASP in 2010.

- **Amazon:** In 2006 ontdekte Chris Shifflet een CSRF kwetsbaarheid op de website van Amazon. Deze had als gevolg dat de aanvaller in naam van het slachtoffer verschillende objecten kon aankopen zonder diens medeweten. Shifflet informeerde Amazon onmiddellijk van dit probleem en deze wist hem te verzekeren het probleem snel aan te pakken. In 2007 had Amazon echter nog steeds niets gedaan om de CSRF kwetsbaarheid weg te werken. Omdat Shifflet zich niet correct behandeld voelde, besloot hij de kwetsbaarheid te delen via het internet. Vervolgens begon Amazon steeds meer last te ondervinden van de kwetsbaarheid en bleven de klachten van klanten niet langer uit. Snel daarna nam Amazon alsnog actie en implementeerde het een CSRF tegenmaatregel. Dit voorbeeld toont aan hoe dit probleem zelfs door grote bedrijven genegeerd werd, terwijl net daar beveiliging een primerend aspect zou moeten zijn [53].

- **Digg:** Digg is een webapplicatie die mensen toelaat bepaalde artikels of websites te delen. In 2006 bevatte de website een kwetsbaarheid die een kwaadaardige website toeliet automatisch een bepaald artikel te *diggen*, m.a.w. te delen met andere mensen dat men dit artikel lezenswaardig vond [2].

- **Google Mail:** Het mailsysteem van Google bevatte in 2007 een CSRF beveiligingsprobleem dat andere websites toegang gaf tot de contacten van een gebruiker. Aanvallers konden deze persoonlijke informatie gebruiken voor marketing doeleinden [12].

- **NYTimes:** In 2008 had de meest gelezen online nieuwsite ter wereld, de New York Times, last van een CSRF kwetsbaarheid. De beveiligingslek gaf een aanvaller de mogelijkheid in het bezit te komen van de adressen van de gebruikers [67].

- **ING Direct:** ING Direct, de vierde grootste bank in de Verenigde Staten, bevatte in 2008 een CSRF kwetsbaarheid waardoor een aanvaller geld kon overplaatsen van de rekening van een slachtoffer naar zijn eigen rekening.

Dit probleem toont aan hoe groot de impact kan zijn op dergelijke gevoelige webtoepassingen [67].

- **MetaFilter:** MetaFilter is een erg populaire weblog met elke maand meer dan 3,5 miljoen unieke bezoekers. In 2008 werd een CSRF exploit ontdekt die een aanvaller toegang gaf tot de gebruikersaccounts. Eerst kon de aanvaller zichzelf vriend maken via de contactlijst. Eens toegevoegd, kon de aanvaller het e-mail adres van een gebruiker veranderen. De aanvaller gebruikte dan dit nieuwe adres om de *Lost Password* functie uit te buiten [67].

- **YouTube:** Youtube, marktleider op gebied van online video's, was het slachtoffer van een CSRF probleem in 2008. Een aanvaller kon onder meer willekeurige video's toevoegen aan de favorieten van een gebruiker, zichzelf toevoegen aan de vriendenlijst en een gebruiker inschrijven op een bepaald kanaal [67].

- **Router probleem - Mexicaanse bank:** In 2008 bevatte een bepaald type van Mexicaanse ADSL routers een CSRF kwetsbaarheid. Een aanvaller zond een kwaadaardige link via e-mail naar het slachtoffer. Eens deze link aangeklikt werd, werd de DNS configuratie veranderd, waardoor de domeinnaam van een populaire Mexicaanse bank plots gelinkt werd aan een kwaadaardig domein. Via deze methode konden aanvallers talrijke authenticatie gegevens bemachtigen van klanten om op die manier zelf in te loggen op de echte website [56].

- **MySpace:** In 2005 werd een CSRF variant in combinatie met een XSS kwetsbaarheid ontdekt op het populaire sociale netwerk MySpace. De worm zorgde ervoor dat de zin "but most of all, Samy is my hero" op het profiel van het slachtoffer verscheen. Binnen een tijdspanne van 20 uren, maakte deze aanval meer dan 1 miljoen slachtoffers. Dit voorbeeld toont aan hoe sterk de combinatie kan zijn van XSS en CSRF kwetsbaarheden binnen hetzelfde domein. Let wel op het feit dat hier geen inter-domein activiteit plaatsvindt en het dus per definitie niet om echte CSRF gaat [52].

2.3 Bestaande beschermingsmaatregelen

Een CSRF beschermingsmaatregel kan zich zowel aan de server-zijde als aan de client-zijde bevinden. In het eerste geval beschermen webapplicaties zichzelf tegen ongeoorloofde aanvallen. Indien alle toepassingen dit zouden doen, zou CSRF geen gevaar mogen vormen. Uit de beschrijving van recente aanvallen in sectie 2.2.4 bleek echter dat een CSRF bescherming aan de *server-zijde* niet vanzelfsprekend mag genomen worden. Vandaar dat ook de client-zijde, m.a.w. de browser, zich best kan beschermen tegen ongeöorloofd inter-domein verkeer.

Aangezien het onderscheid tussen server en client technieken niet enorm groot is en beide partijen vaak samen betrokken zijn bij de tegenmaatregel, maakt sectie 2.3.1 geen specifiek onderscheid tussen client en -servermechanismen. Deze sectie geeft daarentegen een algemeen overzicht van de CSRF beschermingstechnieken die

in de literatuur terug te vinden zijn. Sectie 2.3.2 gaat tot slot dieper in op bestaande browserextensies voor Mozilla Firefox. Het bespreekt de voor- en nadelen van de verschillende mechanismen om tot een grondige vergelijking te komen.

2.3.1 Algemene beschermingstechnieken

Deze eerste sectie bespreekt de algemeen bekende technieken die bescherming kunnen bieden voor CSRF aanvallen. Er wordt vooral gekeken naar de doeltreffendheid en gebruikersimpact van de verschillende maatregelen.

Doeltreffendheid slaat op de volledigheid van de bescherming, terwijl gebruikersimpact rekening houdt met de invloed op de surf-ervaring van de gebruiker. Indien een maatregel alle inter-domein verkeer zou tegenhouden, zou de doeltreffendheid 100% zijn, maar zou de gebruikersimpact erg hoog zijn. Alle verkeer toelaten zou geen invloed hebben op de gebruikerservaring maar zou ook geen CSRF bescherming bieden. Deze twee uiterste manieren om met CSRF om te gaan, houden telkens maar rekening met één criterium. De meeste maatregelen trachten dan ook een goede balans te realiseren tussen gebruikersimpact en doeltreffendheid.

Referer Header

Deze eerste techniek maakt gebruik van de *Referer Header*, een hoofding die browsers doorgaans toevoegen aan elk inter-domein verzoek [19]. De browser vult deze hoofding in met de origine van het verzoek, wat in dit geval het volledige adres van de pagina inclusief protocol, domeinnaam, poort en pad inhoudt. Een webapplicatie kan vervolgens de hoofding controleren en nagaan of hij de afzender vertrouwt.

In theorie lijkt dit mechanisme een voldoende bescherming te bieden, maar in de praktijk komen er enkele problemen aan het licht. Een eerste probleem betreft de manier waarop browsers en proxies omgaan met de *Referer Header*. Omwille van privacy redenen wordt de hoofding namelijk vaak verwijderd door proxy servers en bieden browsers soms de mogelijkheid deze hoofding weg te laten.

Een tweede probleem bevindt zich in het feit dat aanvallers deze hoofding kunnen vervalsen, ook wel *spoofen* genoemd, bijvoorbeeld via Flash plug-ins [31]. Dit wilt zeggen dat een aanvaller zelf de hoofding kan invullen en aldus de server kan misleiden. In de laatste versies van Flash is dit echter niet meer mogelijk aangezien deze het vervalsen van bepaalde velden, waaronder ook de *Referer Header*, verbieden [8].

Het gebruik van de *Origin Header* voorziet een oplossing voor het privacy probleem. Dit systeem werkt op een gelijkaardige wijze op één belangrijk verschil na. Ze beperkt de schending van de privacy tot het protocol, de domeinnaam en de poort van de origine. In tegenstelling tot de *Referer Header* bevat ze dus niet langer het volledige pad van de origine. Hoofdstuk 3 gaat uitgebreid in op het gebruik van deze hoofding door het *Cross-Origin Resource Sharing* mechanisme.

Expliciete herauthenticatie

Een eenvoudige manier om CSRF tegen te gaan, bestaat erin de gebruiker telkens opnieuw zijn authenticatie gegevens te laten invoeren. Ondanks het feit dat deze

manier erg doeltreffend kan zijn, zorgt ze voor een drastisch verminderde gebruikerservaring. Een implementatie van deze techniek in webapplicaties lijkt dus niet erg realistisch. Sommige applicaties maken hiervan gebruik om bepaalde kritische zaken te beschermen, bijvoorbeeld de bevestiging van transacties.

Aangepaste velden in de HTTP hoofding

Het beveiligingsmechanisme voorgesteld door Bart et al. [14] maakt gebruik van aangepaste velden in de hoofding. Deze methode maakt gebruik van de XMLHttpRequest Level 1 (XHR1) API [61], het enige JavaScript of HTML onderdeel van een webpagina dat toelaat zelf gedefinieerde velden toe te voegen aan de hoofding van een verzoek. Zoals reeds vroeger aangehaald is de XHR API beperkt tot het verzenden van verzoeken binnen hetzelfde domein. Een pagina kan op deze manier nagaan of een verzoek vanuit hetzelfde domein afkomstig is door de aanwezigheid van de aangepaste hoofding te controleren. O.a. de JavaScript bibliotheek *prototype.js* en de *Google Web Tool Kit* passen deze techniek toe [21].

Er doen zich echter enkele problemen voor bij dit systeem. Indien een website alsnog een inter-domein verzoek wil toelaten, bestaat er geen manier om dit op een veilige manier te verwezenlijken. Een tweede probleem volgt uit de opkomst van de XMLHttpRequest Level 2 API. De nieuwe versie van deze API laat wel inter-domein verzoeken toe met aangepaste hoofding, wat de bescherming dus verbreekt. Een laatste probleem betreft wederom het vervalsen van de hoofding via een browser plug-in. Net zoals bij het gebruik van de *Referer Header*, loopt men hier het risico dat een aanvaller een Flash plug-in misbruikt om alsnog de aangepaste hoofding te verzenden.

Het gebruik van merktekens

Een belangrijk type CSRF tegenmaatregelen maakt gebruik van merktekens, ook wel *tokens* genoemd. Merktekens zijn willekeurige waarden die een webpagina mee naar de browser verzendt bij een antwoord op een verzoek. Deze waarde kan zich bijvoorbeeld bevinden in een veld van een formulier of als parameter aan een URL toegevoegd worden. Onderstaand voorbeeld maakt gebruik van een merkteken in een formulier:

```
<form action="process.php" method="POST">
    <input name="amount" type="text" value="" />
    <input name="targetAccount" type="text" value="" />
    <input name="CSRFtoken" type="hidden" value="fslkzu7697JnvF4" />
</form>
```

De waarde van de variabele `CSRFtoken` bevat een uniek nummer dat enkel voor de huidige gebruiker geldt. Omwille van de werking van het *Same Origin Policy*, kunnen aanvallers het merkteken niet lezen of veranderen.

Wanneer de browser vervolgens het formulier naar de server verstuurt, zal het verzoek het merkteken bevatten. Op die manier kan de server nagaan of het merkteken

overeenstemt met de voorheen verzonden waarde. Indien de overeenkomst positief is, beschouwt de server het verzoek als legitiem en voert het de bijhorende actie uit.

Hoewel deze aanpak zeer waardevol kan zijn bij de bescherming tegen CSRF aanvallen, zijn er toch enkele bemerkingen. Ten eerste dient de ontwikkelaar de techniek zorgvuldig te implementeren zodat het merkteken niet gemakkelijk te raden valt door malafide gebruikers. Indien men bijvoorbeeld een systeem hanteert waarbij de gebruikersnaam als unieke waarde fungeert, zal een andere gebruiker deze gemakkelijk kunnen raden en zelf een legitiem verzoek kunnen construeren.

Ten tweede moet ook gezorgd worden dat het merkteken niet op één of andere manier in handen kan komen van andere gebruikers. Zo kunnen bijvoorbeeld XSS kwetsbaarheden misbruikt worden om een merkteken te bemachtigen. Een aanvaller injecteert een script in het domein van een website dat vervolgens toegang heeft tot het volledige DOM model van de website en het merkteken zo simpelweg kan opvragen.

Enkele voorbeelden van applicaties die een vorm van deze techniek gebruiken:

- *NoForge:* NoForge is een proxy aan de server-zijde waarlangs al het verkeer tussen de browser en de server passeert [29]. NoForge voegt een sessie-afhankelijk merkteken toe aan de aanwezige URL's in elk uitgaand bericht en controleert elke inkomend verzoek van de browser op aanwezigheid en correctheid van deze waarde. Indien de proxy na vergelijking van beide merktekens een overeenkomst vindt, zal het verzoek worden toegelaten. Het gebruik van een proxy houdt echter wel een specifiek nadeel in. Links die dynamisch gecreëerd worden in de browser zijn namelijk niet beschermd aangezien de proxy ze niet kan herkennen.

- *CSRFGuard en CSRFx:* CSRFGuard is een CSRF bescherming voor Java EE applicaties die eveneens gebruik maakt van sessie-afhankelijke merktekens [45]. Het systeem is geïmplementeerd als een JEE filter en voegt aan elke uitgaande link een merkteken toe dat bij ontvangst gecontroleerd wordt op aanwezigheid en correctheid. CSRFx maakt gebruik van een gelijkaardig systeem en bestaat uit een bibliotheek waarvan PHP applicaties gebruik kunnen maken [27]. Deze beschermingsmaatregelen zijn rechtstreeks ingebouwd in de ontwikkelingsomgeving waardoor ze ook dynamisch gecreëerde links kunnen beschermen.

2.3.2 Browser extensies

Zoals sectie 2.2.4 reeds aanhaalde, zijn heel wat websites kwetsbaar voor CSRF aanvallen waardoor de gebruiker best zelf een beschermingsmaatregel installeert. Een browser extensie heeft het voordeel dat een volledig beeld beschikbaar is van de context waarin een verzoek zich voordoet. Op deze manier kan de extensie een beslissing nemen op basis van de verzamelde informatie binnen de browser om deze tot slot op het uitgaand verzoek af te dwingen.

Elke onderzochte tool in deze sectie werd ontwikkeld voor Mozilla Firefox. Omwille van het gebrek aan een robuust extensie platform zijn tot op heden nog geen CSRF extensies teruggevonden op andere browsers [18].

	GET		POST
	HTTP	HTTPS	
Sessie cookies	Niet gevoelig	Gevoelig	Gevoelig
Persistente cookies	Niet gevoelig		Gevoelig
HTTP Authenticatie	Gevoelig		

Tabel 2.2: Het standaard beleid dat BEAP toepast [34].

RequestPolicy

RequestPolicy voorkomt CSRF aanvallen door alle inter-domein verkeer te blokkeren waarvoor de gebruiker zelf niet direct verantwoordelijk is [49]. Bij blokkering kan een gebruiker eventueel een exceptie stellen die het verzoek alsnog toelaat. Het blokkerend gedrag van deze extensie zorgt voor een sterke belemmering van de gebruikerservaring. Zo functioneert het openen van een zoekresultaat op Google niet correct, omdat een klik op een link de gebruiker eerst verwijst naar een andere pagina die de gebruiker dan doorstuurt naar de opgevraagde website [48]. RequestPolicy is één van de weinige maatregelen waarvoor een mobiele variant beschikbaar is. Deze bevat dezelfde functionaliteit waarvan de gebruiksinterface van de geteste versie spijtig genoeg nog niet was aangepast aan de kleinere resolutie.

BEAP (AntiCSRF)

AntiCSRF, een tweede maatregel aan de client-zijde, maakt gebruik van BEAP, wat staat voor *Browser-Enforced Authenticity Protection* [34]. Tabel 2.2 geeft een overzicht van het beleid dat BEAP toepast. In plaats van inter-domein verkeer te blokkeren, implementeert het een verfijnder beleid om tot een beslissing te komen. Deze beslissing houdt ofwel de toelating van een verzoek in ofwel de verwijdering van sessie informatie. De gebruikersimpact van BEAP is bijgevolg minder groot als die van RequestPolicy. Een belangrijke opmerking dient gemaakt te worden over de manier waarop BEAP de `GET` methode behandelt. Indien HTTP het gebruikte protocol is, verwijdert de extensie namelijk geen cookies uit een verzoek. BEAP geeft als argument voor deze aanpassing dat de werking van bepaalde web 2.0 applicaties zo minder aangetast wordt. Ondanks de verbeterde gebruikservaring loopt men hier toch het risico dat dit type verzoek gebruikt wordt om bepaalde acties tot stand te brengen, waardoor het risico op succesvolle CSRF aanvallen groot blijft.

CSRF protector

Uit het onderzoek van Zeller et al. volgde de browser extensie CSRF Protector[67]. Een eerste stap in dit systeem gaat na wat de methode is van het verzoek. Indien het verzoek een andere methode gebruikt als de `POST` methode, laat CSRF Protector het verzoek sowieso toe. In een volgende stap laat de extensie elk verzoek toe dat

volgens het SOP niet voldoet aan de voorwaarden van inter-domein verzoek. Tot slot kijkt de extensie naar de eventuele aanwezigheid van een *Adobe Cross-Domain Policy* (ACDP) op de bestemmende server [7]. Deze file, `crossdomain.xml` genoemd, wordt gebruikt door Flash plug-ins om eventueel inter-domein verkeer mogelijk te maken. CSRF Protector laat een verzoek toe indien de eigenschappen overeenkomen met een regel uit het ACDP. Indien aan geen van de voorgaande voorwaarden voldaan is, vraagt de extensie aan de gebruiker of hij een uitzondering wil maken voor dit verzoek. Als dit niet het geval is, blokkeert CSRF Protector het betreffende verzoek.

Onderstaand voorbeeld bevat code van een mogelijke `crossdomain.xml` file:

```
<cross-domain-policy>
    <allow-access-from domain="foo.example.com" />
</cross-domain-policy>
```

In dit geval zou het domein waarop deze file zich bevindt, toelaten om verzoeken van `foo.example.com` te ontvangen.

Zoals reeds bleek in sectie 2.2.1 over de verschillende CSRF aanvalsvectoren, is het negeren van `GET` verzoeken geen goed idee. Ondanks dat de officieële specificatie aanhaalt dat webpagina's de `GET` methode niet mogen gebruiken voor toestand wijzigende acties, blijken vele ontwikkelaars dit te negeren [48]. Verder houdt het gebruik van een lijst van uitzonderingen, opgebouwd door de gebruiker, ook enkele risico's in. Er kan namelijk niet verwacht worden dat alle gebruikers even accurate beslissingen kunnen maken over de veiligheid van inter-domein verkeer.

NoScript

De standaard werking van NoScript schakelt verschillende elementen van een webpagina uit [6]. Zo worden JavaScript, Java, Flash en Silverlight objecten niet opgestart omdat ze eventuele veiligheidsrisico's kunnen bevatten.

Een specifieke functie van NoScript dat als CSRF afweermechanisme dienst kan doen, is de *Application Boundaries Enforcer* (ABE). Via deze tool kunnen gebruikers specifieke regels instellen die betrekking hebben op het inter-domein verkeer. Standaard staan volgende regels vermeld:

```
Site LOCAL
Accept from LOCAL
Deny
```

De voorgaande regel geeft aan dat sites enkel verzoeken mogen accepteren van hun eigen domein. Dit standaard beleid blokkeert inter-domein verkeer volledig waardoor de gebruiker beschermd is tegen CSRF aanvallen, maar waardoor de gebruikerservaring drastisch achteruit gaat. ABE biedt echter nog verfijndere mogelijkheden zoals het verwijderen van cookies en HTTP authenticatie informatie van inter-domein verkeer. Onderstaande ABE regel bewerkstelligt dit voor alle inter-domein verzoeken die gebruik maken van de `GET` of `POST` methode:

```
Site *
Anonymize GET POST
```

NoScript biedt net zoals RequestPolicy een mobiele variant aan voor Mozilla Fennec. Deze is echter wel beperkt tot enkele standaard beveiligingsniveau's waaruit de gebruiker kan kiezen zonder de aanwezigheid van de ABE module. Na enkele tests op elk beveiligingsniveau bleek een eventuele CSRF bescherming zelfs volledig afwezig te zijn.

Content gebaseerde CSRF detectie

Het werk van Shahriar en Zulkernine hanteert een volledig andere aanpak dan de hiervoor beschreven extensies [51]. In plaats van te werken met een lijst van uitzonderingen of een beleid op basis van specifieke eigenschappen van een verzoek, houdt de extensie rekening met de inhoud van een webpagina. Op basis van bepaalde eigenschappen van een pagina tracht de extensie na te gaan of een bepaald inter-domein verzoek al dan niet gevaarlijk is. Zo controleert de extensie bijvoorbeeld of er wel degelijk een afbeelding wordt ingeladen bij het gebruik van de `img` tag.

CsFire

CsFire is een extensie die voortbouwt op het werk van Maes et al. [33]. Tabel 2.3 stelt het beleid voor waarvan CsFire gebruik maakt [48]. Gelijkaardig aan BEAP, implementeert CsFire een aanpak die zoveel mogelijk web 2.0 functionaliteit wil toelaten door verzoeken niet te blokkeren, maar confidentiële sessie informatie te verwijderen. Het beleid is daarenboven gebaseerd op het uitgebreid onderzoek bij 15 gebruikers dat reeds in de inleiding aan bod kwam. Een belangrijk gevolg dat hieruit voortvloeide was de opmerking dat de gebruiker zelf heel weinig inter-domein verkeer initieert en dat de `GET` methode veruit de meest gebruikte is. CsFire kiest er dan ook voor `GET` verzoeken zonder parameters rechtsreeks toe te laten, indien de gebruiker ze expliciet veroorzaakt. De extensie laat de overige inter-domein verzoeken toe, maar verwijdert eventueel aanwezige cookies of HTTP authenticatie informatie. Tot slot maakt CsFire nog gebruik van extra regels die de extensie regelmatig van de officiële CsFire server downloadt. Deze regels bevatten een aantal uitzonderingen op de normale werking van CsFire die het beslissingsalgoritme in rekening brengt. Daarnaast kan ook de gebruiker gelijkaardige regels toevoegen aan een lokale lijst van uitzonderingen.

2.3.3 Blijvende onvolkomenheden

Uit het voorgaande blijkt dat een CSRF maatregel ontwikkelen geen sinecure is. Deze sectie vat de belangrijkste conclusies bondig samen.

- **Balans tussen doeltreffendheid en gebruikerservaring**

  Wat betreft de impact op de gebruikerservaring scoren vooral CsFire en BEAP goede resultaten. Ze verwijderen enkel de sessie informatie en blokkeren geen

Eigenschappen			Beslissing
GET	Parameters		STRIP
	Geen parameters	Gebruiker	AANVAARD
		Systeem	STRIP
POST	Gebruiker		STRIP
	Systeem		STRIP

Tabel 2.3: Het standaard beleid van CsFire

inter-domein verkeer zoals CSRF Protector en RequestPolicy. Vooral CsFire bevat een goede balans die voortvloeide uit een onderzoek naar inter-domein verkeer bij 15 proefpersonen. Het standaard beleid van NoScript blokkeert alle inter-domein verzoeken, al kan de gebruiker dit wel aanpassen zodat een gelijkaardig beleid als bij CsFire in werking treedt.

- **De verantwoordelijkheid van de gebruiker**

  Sommige maatregelen leggen erg veel verantwoordelijkheid bij de gebruiker door te vragen of een bepaald inter-domein verzoek toegelaten mag worden. Dit is enerzijds niet meer haalbaar door het stijgend aantal inter-domein verzoeken, anderzijds kan zelfs de meest ervaren gebruiker misschien niet altijd de juiste beslissing nemen. Ook hier scoort CsFire weer beter dan de meeste andere extensies, aangezien het standaard beleid volledig transparant te werk gaat.

- **Gebrek aan beschermingsmaatregelen voor mobiele apparaten**

  Het gedrag van browsers op mobiele apparaten lijkt sterk op dat van desktop browsers, waardoor ook hier dezelfde CSRF risico's van tel zijn. Dwivedi, Clarck en Thiel merken in hun werk op dat mobiele gebruikers minsten even geneigd zijn om bepaalde gevaarlijke websites te bezoeken na het bezoek van een gevoelige applicatie (bv. bank website) [17]. Verder beweren ze ook dat mobiele gebruikers sneller een bepaalde link zouden aanklikken in een e-mail terwijl er een browser sessie actief is. Bedrijven, waaronder ook banken, beginnen ook meer en mee mobiele applicaties aan te bieden wat de mogelijke schade door CSRF aanvallen nog kan vergroten [20].

- **Interactie tussen browsers en domeinen**

  Maar weinig extensies bevatten een systeem waarbij de browser en de bestemmende server communiceren welke domeinen toegang mogen krijgen tot de server. Enkel CSRF protector maakt gebruik van het eventueel aanwezige *Adobe Cross-Domain policy* op de server. Indien dit beschikbaar is, voegt de extensie de uitzonderingen dynamisch toe zonder dat de gebruiker gestoord wordt. CsFire voorziet geen directe interactie met de bestemmende server maar maakt wel gebruik van extra downloadbare regels van de CsFire website. CsFire kan zo regelmatig uitzonderingen toevoegen aan het beslissingsalgoritme.

Dit hoofdstuk gaf een uitgebreide verklaring van het begrip *Cross-Site Request Forgery* en besteedde zowel aandacht aan de werking, gevolgen en beschermingsmogelijkheden voor dit type aanvallen. Terwijl CSRF een bedreiging is voor veilige inter-domein communicatie, zijn er de laatste jaren technieken ontwikkeld met als doel dit soort communicatie op een veiligere manier mogelijk te maken. Hoofdstuk 3 gaat daarom uitgebreid in op enkele oudere en recentere technieken die dit doel voor ogen hebben.

Hoofdstuk 3

Mechanismen voor inter-domein communicatie

3.1 Traditionele methoden

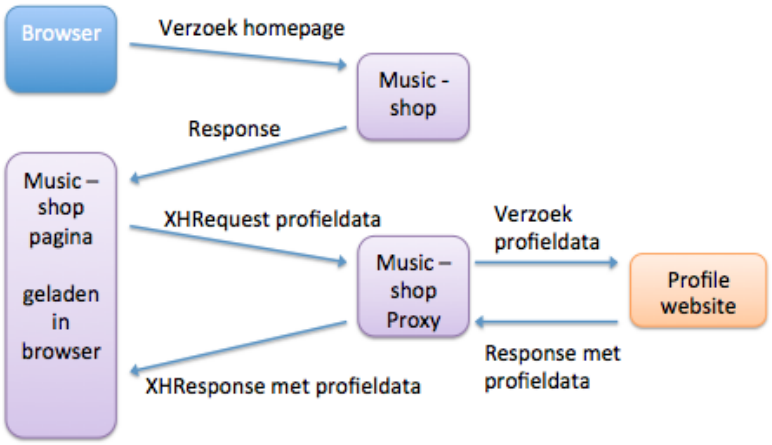
Zoals sectie 2.1.1 aanhaalde, beperkt het *Same Origin Policy* de inter-domein mogelijkheden binnen de browser. Dit systeem is niet expliciet voorzien van een mechanisme om communicatie tussen verschillende domeinen te faciliteren. Omwille van de nood aan dit soort communicatie ontstonden binnen de grenzen van het SOP alsnog methodes om dit te realiseren. Deze sectie bespreekt enkele van deze traditionele ontwikkelingen [47].

3.1.1 XMLHttpRequest Proxies

Een eerste methode maakt gebruik van het XMLHttpRequest Level 1 object, een onderdeel van de JavaScript API dat sectie 2.2.1 reeds toelichtte. Deze XHR verzoeken zijn beperkt omdat webpagina's ze enkel binnen het eigen domein kunnen versturen. Om alsnog inter-domein verkeer mogelijk te maken, maken websites gebruik van een proxy aan de server-zijde.

Figuur 3.1 geeft schematisch weer hoe dit in zijn werk gaat. In een eerste stap kiest een gebruiker ervoor de site te bekijken van een muziekwinkel. Nadat de browser de homepagina van de webshop heeft ingeladen tracht de webshop de inhoud van de pagina te personaliseren door gebruikersvoorkeuren van een externe profielwebsite af te halen. Daarom verstuurt de webpagina een XMLHttpRequest naar de proxy aan de server-zijde. Aangezien er in dit geval een gebruikersnaam vereist is en eventueel ook een paswoord indien de gegevens niet vrij beschikbaar zijn, is de gebruiker verplicht deze gegevens mee te delen aan de muziekwinkel. In de derde stap kan de proxy de informatie ophalen van de profielwebsite om de verkregen profieldata weer door te spelen naar de browser. Tot slot gebruikt de muziekwinkel deze profielgegevens om de website aan te passen aan de smaak van de gebruiker.

Dit voorbeeld wijst reeds op het gevaar dat deze techniek met zich meebrengt. Een geauthenticeerde sessie vereist het volledig vertrouwen van de gebruiker in de



Figuur 3.1: Een inter-domein scenario dat gebruik maakt van een XHR proxy op de server.

server waar de proxy zich bevindt aangezien de browser de login informatie via dit kanaal verzendt. Voor niet-kritische inter-domein communicatie biedt deze methode echter wel interessante mogelijkheden om het SOP te omzeilen.

3.1.2 JSON-Padding

Het SOP verbiedt normalerwijze het inlezen en veranderen van gegevens tussen websites van verschillende domeinen. Deze restrictie bestaat echter niet wanneer een website een script inlaadt, bijvoorbeeld via volgende code:

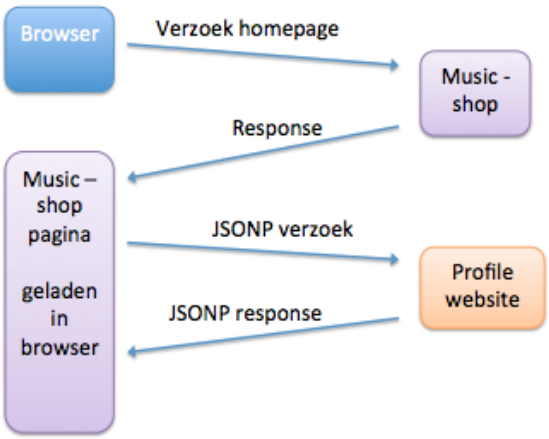
```
<script src="http://foo.example.com/script.js" />
```

Dit script maakt in deze context integraal deel uit van het domein dat het script heeft ingeladen. *JSON with padding*, of JSONP, is een techniek die op deze wijze inter-domein communicatie mogelijk maakt [4]. JSON verwijst naar een gestandaardiseerd dataformaat dat websites gebruiken om informatie uit te wisselen [5]. Het scenario in figuur 3.2 maakt gebruik van JSONP en omvat volgende stappen.

Vooreerst laadt de browser de homepagina van de muziekwinkel. Deze pagina bevat volgend script dat een inter-domein verzoek verstuurt naar een profielwebsite.

```
<script src="http://profileWebsite.com/script.js?callback=parseJson" />
```

Op het moment van inladen bestaat er reeds een actieve sessie met de profielwebsite waarbij de browser een cookie heeft opgeslagen met een uniek identificatienummer voor de gebruiker. De browser verzendt deze cookie impliciet mee bij elk verzoek naar de profielwebsite, waardoor deze weet om welke gebruiker het gaat. De link naar het script krijgt de parameter `callback=parseJson` mee die aangeeft dat de functie



FIGUUR 3.2: Een inter-domein scenario dat gebruik maakt van de *JSON with padding* techniek.

`parseJson` beschikbaar is voor de verwerking van het antwoord. De profielwebsite leest deze waarde in en gebruikt ze bij de opstelling van het script. Het script in het antwoord van de profielwebsite kan er dan als volgt uitzien:

```
parseJson({"Favorite style": "Jazz", "Favorite artist" : "Miles Davis"});
```

Het script roept de functie `parseJson` op en gebruikt de verkregen informatie om de inhoud van de webpagina aan te passen.

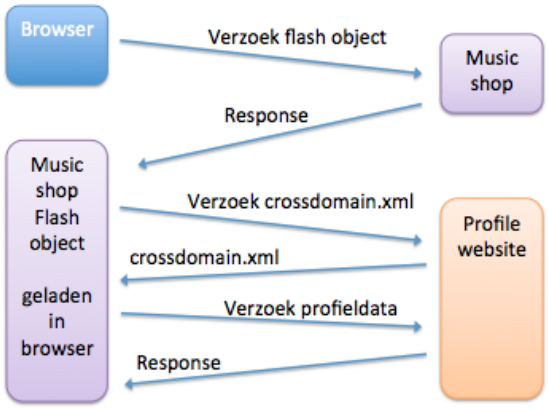
Enkele veiligheidsproblemen dient men in acht te nemen bij het gebruik van deze techniek. Enerzijds moet de muziekwinkel het script van de profielwebsite volledig vertrouwen aangezien het toegang tot het ganse domein heeft. Anderzijds loopt deze techniek ook gevaar voor *Cross-Site Request Forgery*, een beveiligingsprobleem dat hoofdstuk 2 reeds uitvoerig behandelde. De profielwebsite moet kunnen nagaan welke website het verzoek uitvoerde opdat niet elke website deze vertrouwelijke gegevens zou kunnen verkrijgen.

3.1.3 Plug-ins

Wanneer een website een plug-in laadt, is deze niet onderhevig aan het *Same Origin Policy* van de browser maar aan eigen ontworpen beleidsregels [66]. Deze sectie bespreekt enkele frequent gebruikte plug-ins die op deze manier inter-domein communicatie toelaten.

- **Adobe Flash**

  De bespreking van CSRF Protector in sectie 2.3.2 lichtte reeds toe hoe Adobe Flash plug-ins inter-domein verkeer toelaten. Flash maakt gebruik van een `crossdomain.xml` bestand op de server dat de inter-domein mogelijkheden voor een bepaalde server bepaalt. Figuur 3.3 bevat de verschillende stappen.



Figuur 3.3: Een Adobe Flash object dat een inter-domein verzoek uitvoert.

Een eerste stap bestaat uit het inladen van het Flash object. Adobe Flash beschouwt de website waarop het Flash object zich bevindt als origine van het object, in dit geval de website van de muziekwinkel. In een tweede stap tracht het Flash object gegevens in te laden van de profielwebsite. Opdat dit verzoek kan gebeuren moet de plug-in eerst het `crossdomain.xml` bestand downloaden met daarin de toegelaten domeinen. Dit kan er als volgt uitzien:

```
<cross-domain-policy>
    <allow-access-from domain="musicshop.com" />
</cross-domain-policy>
```

De bovenstaande regel geeft aan dat inter-domein communicatie in dit geval toegelaten is. Een laatste stap bestaat uit het zenden van het werkelijke verzoek voor profielinformatie waarbij de identificatie van de gebruiker verloopt via de impliciet mee verzonden cookies. Tot slot kan het Flash object de profielinformatie gebruiken om de inhoud aan te passen naar de voorkeuren van de gebruiker.

- **Microsoft Silverlight**

  Silverlight is een tegenhanger van Flash, ontwikkeld door Microsoft, en heeft eveneens mogelijkheden om inter-domein communicatie mogelijk te maken [37]. Silverlight laat echter zowel het gebruik van Adobe's `crossdomain.xml` bestand toe als het gebruik van het `clientaccesspolicy.xml` bestand. Dit laatste bestand bevindt zich eveneens op de server en kan er als volgt uitzien:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
```

```
<cross-domain-access>
  <policy>
    <allow-from http-request-headers="*">
      <domain uri="http://musicshop.com"/>
    </allow-from>
    <grant-to>
      <resource path="/profileInfo/" include-subpaths="true"/>
    </grant-to>
  </policy>
</cross-domain-access>
</access-policy>
```

Bovenstaand regels geven de mogelijkheid om bepaalde gegevens van de profiel-website te delen met de muziekwinkel. In tegenstelling tot het `crossdomain.xml` bestand kan `clientaccesspolicy.xml` ook aangeven welke paden toegankelijk zijn.

Tot slot geeft deze sectie nog enkele opmerkingen bij het gebruik van deze plug-ins. Ten eerste kan men niet verwachten dat alle browsers plug-ins ondersteunen. Zo is de ondersteuning van Flash op heel wat mobiele apparaten erg gebrekkig [36]. Verder heeft het verleden reeds geduid op de gevaren die plug-ins met zich meebrengen [47]. Veiligheidsupdates dringen zich dan ook regelmatig op, om bepaalde kwetsbaarheden weg te werken. Recentelijk bracht Adobe nog een update uit om een bepaalde CSRF kwetsbaarheid in Flash objecten weg te nemen [3].

3.2 Recente ontwikkelingen

Deze sectie bespreekt twee recent ontwikkelde mechanismen die rekening trachten te houden met de specifieke veiligheidsvereisten van inter-domein communicatie. Sectie 3.2.1 behandelt *Cross-Origin Resource Sharing* en sectie 3.2.2 het *Uniform Messaging Policy*, twee specificaties die sterk verschillen in hun designfilosofie.

3.2.1 Cross-Origin Resource Sharing

Aanpak

CORS, kort voor *Cross-Origin Resource Sharing* [59], is een mechanisme dat als doel heeft inter-domein gegevens op een veilige manier te kunnen delen. Het bouwt voort op de huidige werking van het SOP en houdt rekening met de bestaande sessie mechanismen beschreven in sectie 2.1.2 [58]. De hoofdgedachte bestaat erin dat elk inter-domein verzoek een *Origin Header* [13] moet bevatten. Gelijkaardig aan de *Referer Header* bevat deze hoofding de origine van een verzoek maar blijft ze in dit geval beperkt tot het domein, protocol en poortnummer van de verzender. Dit om de schending van de privacy te vermijden. Elke pagina kan een toegangslijst krijgen waarin zich de domeinen bevinden die toegang mogen krijgen tot de data op een

bepaalde pagina. Op deze manier kan een webtoepassing de toegang weigeren tot data of er zelfs voor zorgen dat de browser een verzoek niet verzendt.

CORS maakt een onderscheid tussen verzoeken op basis van de gebruikte methode, het type inhoud en de gebruikte velden in de hoofding. Wanneer een verzoek voldoet aan bepaalde beperkingen beschouwt CORS het als een eenvoudig verzoek, of *simple request*. Indien een verzoek daarentegen niet voldoet aan deze vereisten, verzendt het CORS mechanisme eerst een voorafgaand verzoek, of *preflight request*. De browser verzendt deze vóór het eigenlijke verzoek om bij de server na te gaan deze wel effectief mag verzonden worden. Omdat de W3C specificatie[59] geen duidelijke taxonomie voorziet voor niet-eenvoudige verzoeken, zal de rest van deze masterproef de term geavanceerd verzoek, of *advanced request* hanteren.

- *Een eenvoudig verzoek*

  De voorwaarden voor een eenvoudig verzoek zijn als volgt:

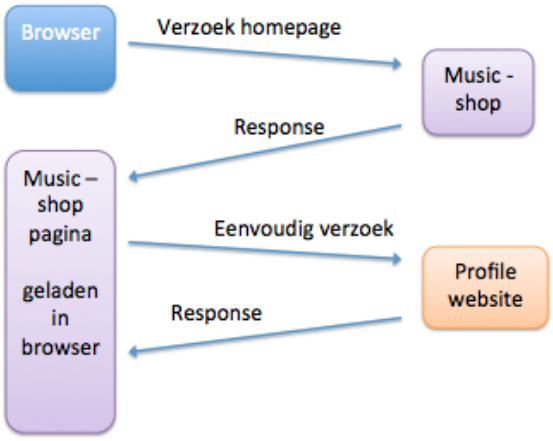
  - De methode van het verzoek moet een hoofdletter-gevoelige overeenkomst zijn met `GET`, `POST` of `HEAD`.
  - Elk veld van de hoofding van een verzoek moet een ASCII hoofdletter-gevoelige overeenkomst zijn met `Accept`, `Accept-Language`, `Content-Language`, `Last-Event-ID` of `Content-Type`.
  - De waarde van de *Content-Language* hoofding moet een ASCII hoofdletter-gevoelige overeenkomst zijn met `application/x-www-form-urlencoded`, `multipart/form-data` of `text/plain`.

  Een browser verzendt een eenvoudig verzoek naar de server met of zonder eventuele opgeslagen sessie gegevens. Een ontwikkelaar kan dit expliciet bepalen via de `withCredentials` optie. Na de verzending van een verzoek antwoordt de server met een bericht waarin het resultaat van het verzoek vervat zit. Daarnaast bevat het ook velden in de hoofding die aanduiden of het ontvangende domein toegang mag krijgen tot dit antwoord. Op deze manier biedt het CORS mechanisme de mogelijkheid om uit het standaard *Same Origin Policy* te stappen en op die manier inter-domein gegevens uit te wisselen. Volgende sectie voorziet een voorbeeldscenario dat een eenvoudig verzoek illustreert.

- *Een geavanceerd verzoek*

  Indien een verzoek niet voldoet aan de voorwaarden van een eenvoudig verzoek, behandelt het CORS mechanisme het als een geavanceerd verzoek. Een dergelijk verzoek bestaat uit twee opeenvolgende stappen.

  In een eerste stap verzendt de browser een voorafgaand verzoek naar de bestemmende server met behulp van de `OPTIONS` methode. Dit bericht bevat informatie over het eigenlijke verzoek dat een bepaalde website wil verzenden. Het bevat de namen van de aangepaste velden in de hoofding, de methode en het eventueel gebruik van sessie informatie maar mag in geen geval zelf authenticatie informatie met zich meedragen. Vervolgens stuurt de server een



FIGUUR 3.4: Een scenario van een eenvoudig CORS verzoek.

antwoord naar de browser waarin de toegelaten methodes, velden in de hoofding en de eventuele toelating voor sessie informatie zich bevindt.

In een tweede stap verzendt de browser het eigenlijke verzoek naar de server, maar enkel indien deze de voorwaarden uit het voorafgaande verzoek niet schendt.

Het verschil tussen beide types verzoeken is redelijk sterk. Terwijl een eenvoudig verzoek enkel een mechanisme voorziet om uit het *Same Origin Policy* te stappen, kan een geavanceerd verzoek de uitvoer van een bepaald verzoek volledig verbieden.

Voorbeeld scenario's

- *Eenvoudig verzoek scenario*

  Figuur 3.4 geeft een overzicht van het eenvoudig verzoek scenario. Het CORS mechanisme treedt in werk wanneer de browser de website van de online muziekwinkel inlaadt. Deze website zendt een eenvoudig verzoek naar de profielwebsite met de `withCredentials` optie op `true`. Aangezien de gebruiker reeds ingelogd was op de profielwebsite vooraleer hij de muziekwebsite opende, verzendt de browser de opgeslagen sessie informatie mee met het verzoek. De browser voegt ook de *Origin Header* toe aan het verzoek, met `http://musicshop.com` als waarde.

  De profielwebsite antwoordt op zijn beurt met de profielinformatie die de muziekwinkel opvroeg, nadat zowel de *Origin Header* als de sessie informatie gevalideerd werd. De server voegt volgende CORS gerelateerde velden toe aan de hoofding van het antwoord:

- **Access-Control-Allow-Origin:** Dit veld bevat de waarde `http://musicshop.com` waardoor de muziekwinkel toegang krijgt tot de data.
- **Access-Control-Allow-Credentials:** Dit veld bevat de waarde `true` wat aangeeft dat de muziekwinkel toegang krijgt tot de data bij gebruik van sessie informatie.

- *Geavanceerd verzoek scenario*

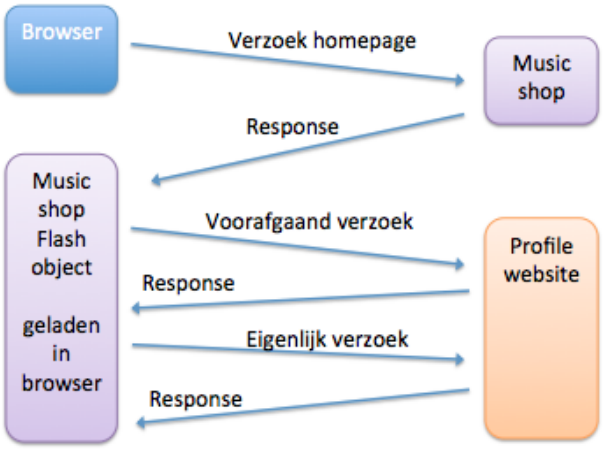
  Het geavanceerd verzoek scenario is wat ingewikkelder dan het eenvoudig verzoek scenario aangezien het gaat om een tweeledig verzoek. Figuur 3.5 toont de verschillende stappen van een geavanceerd verzoek.

  In een eerste stap verzendt de browser een voorafgaand verzoek naar de server. Een belangrijke opmerking hierbij is dat deze nooit sessie informatie meedraagt, dit kan enkel gebeuren bij de verzending van het eigenlijke verzoek. De volgende CORS velden worden toegevoegd aan de hoofding van het verzoek:

  - **Origin Header:** Dit veld bevat de origine van het verzoek, `http://musicshop.com`.
  - **Access-Control-Request-Method:** Dit veld bevat de gebruikte methode in het eigenlijke verzoek, in dit geval de `GET` methode.
  - **Access-Control-Request-Headers:** Dit veld bevat de eventuele aangepaste velden in hoofding van het eigenlijke verzoek. In dit geval vraagt de webpagina het gebruik van de `Special_Header` hoofding aan.

  De profielwebsite antwoordt met een bericht met volgende velden in de hoofding:

  - **Access-Control-Allow-Origin:** Dit veld bevat de waarde `http://musicshop.com` wat aangeeft dat de browser het resultaat mag delen met de muziekwinkel.
  - **Access-Control-Allow-Methods:** Dit veld voorziet een lijst van de toegelaten methodes, in dit geval `GET, POST, PUT` en `DELETE`, al is de website niet verplicht `GET` en `POST` expliciet in te vullen. Deze zijn namelijk altijd toegelaten.
  - **Access-Control-Allow-Headers:** Dit veld bevat de waarde `Special_Header` wat aangeeft dat de server deze aangepaste hoofding toelaat.
  - **Access-Control-Allow-Credentials:** Dit veld bevat de waarde `true`, waardoor sessie informatie toegelaten is.
  - **Access-Control-Max-Age:** Dit veld bepaalt de maximale tijd dat een de verkregen informatie uit het voorafgaand verzoek bewaart mag blijven in de browser. Dit vermijdt een overbelasting van identieke voorafgaande verzoeken.



FIGUUR 3.5: Een scenario van een geavanceerd CORS verzoek.

– **Access-Control-Expose-Headers:** Dit voorbeeld maakt geen gebruik van dit veld waardoor deze geen waarde bevat. Normaal gezien bevinden zich hier de namen van de velden in de hoofding die andere domeinen mogen lezen.

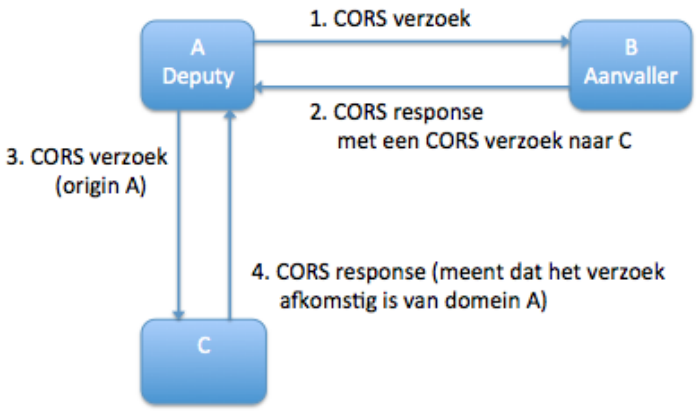
Vervolgens verzendt de browser het eigenlijke verzoek, maar enkel indien voldaan is aan de voorwaarden van het voorafgaand verzoek. In dit geval voldoet het verzoek aan alle voorwaarden waardoor de browser het eigenlijke verzoek naar de server verzendt, inclusief een *Origin Header* en de opgeslagen sessie informatie.

Tot slot stuurt de server een antwoord met daarin de opgevraagde informatie en gepaste CORS hoofding, waardoor de website van de muziekwinkel deze de profielinformatie kan gebruiken om de website te personaliseren.

Browser ondersteuning

De meeste grote browser ontwikkelaars ondersteunen een bepaalde variant van de CORS specificatie. De ontwikkelaars van Mozilla Firefox, Safari en Google Chrome beweren allen CORS te ondersteunen volgens de W3C specificatie [55, 38, 46]. Deze browsers maken gebruik van een nieuwe versie van de XMLHttpRequest API (XHR). Daar waar de XHR Level 1 API enkel verzoeken toelaat binnen hetzelfde domein, biedt de XHR Level 2 API ook inter-domein communicatie mogelijkheden aan, zoals beschreven in de CORS specificatie.

Microsoft voorziet een gedeeltelijke implementatie van de CORS specificatie in Internet Explorer 8 en 9 door middel van de XDomainRequest API (XDR). Deze andere aanpak volgt uit het geloof dat CORS niet volledig veilig is [35]. Enkele verschillen zijn onder andere de beperking tot `GET` en `POST` methodes en het onvermogen



Figuur 3.6: Een schema van een CSRF-achtige aanval in een CORS situatie

om sessie informatie en aangepaste velden in de hoofding te verzenden. De volgende sectie betreffende CORS problemen ondersteunt deze vermoedens aangezien CORS te kampen heeft met enkele veiligheidsproblemen.

Problemen

Ondanks dat de CORS specificatie veilige deling van inter-domein gegevens beoogt, blijven toch enkele problemen bestaan [58]:

- *Cross-Site Request Forgery* [42]

  Cross-Site Request Forgery blijft een probleem omwille van twee redenen. Ten eerste voorziet een eenvoudig CORS verzoek geen impliciete bescherming tegen vervalste verzoeken. De CORS API laat namelijk toe om een eenvoudig verzoek sowieso te verzenden, zonder voorafgaande controle. Enkel het antwoord op een verzoek wordt beschermd tegen het inlezen of de aanpassing door andere domeinen. Volgend voorbeeld verduidelijkt dit probleem. Indien een website profiel informatie aanbiedt maar de server geen CORS ondersteuning geniet, zal deze het verzoek sowieso uitvoeren en het resultaat teruggegeven aan de browser. Het andere domein zal het antwoord echter niet kunnen inlezen en gebruiken om de website te personaliseren aangezien de server dit niet toeliet via de gepaste CORS hoofding. Veronderstel nu dat diezelfde profielwebsite ook de mogelijkheid voorziet om informatie te verwijderen via een verzoek. Dan zal een eenvoudig verzoek dat aankomt bij de server deze operatie succesvol uitvoeren. Geavanceerde verzoeken zijn daarentegen wel beschermd, aangezien de browser het eigenlijke verzoek pas verzendt na expliciete toelating via een voorafgaand verzoek.
  Een tweede CSRF kwetsbaarheid komt aan het licht wanneer een bepaalde website A informatie ontvangt van een kwaadaardige website B en deze informatie

in zijn eigen domein laadt. Figuur 3.6 biedt een schematisch overzicht van deze situatie. Website A bevindt zich zowel in de toegangslijst van website B als website C. Stel dat website B eigendom is van een aanvaller die beseft dat de informatie die hij stuurt naar website A, geladen wordt in het domein van website A. Website B kan dan informatie versturen naar website A waardoor deze een inter-domein verzoek verstuurt naar website C, die onterecht zal geloven dat dit verzoek geïnitieerd werd door website A. Websites kunnen dit zogenaamd *confused deputy* probleem vermijden door zich te houden aan een goed design, met name het *Don't Be A Deputy* design (DBAD) [54]. Indien websites dit design toepassen, zal website A nooit toelaten om informatie verkregen van een andere website, in zijn eigen naam uit te voeren.

- *Onafdwingbaar veiligheidsbeleid*

  Een derde probleem betreft het ambigue gedrag van *JSON* bestanden [58]. Zoals sectie 3.1 reeds aangaf, gebruiken sommige web toepassingen de *JavaScript Object Notation* [4] om gegevens te delen via de `<script>` tag. Websites kunnen JSON bestanden inladen in het script object die vervolgens toegankelijk zijn voor een website, ongeacht de aanwezigheid van meegegeven CORS richtlijnen.

- *XMLHttpRequest Level 1 compatibiliteit*

  Een volgende probleem bevindt zich in het veranderde gedrag van de XMLHttpRequest API. De XHR Level 1 API [61] is beperkt door het feit dat verzoeken enkel binnen éénzelfde domein kunnen gebeuren terwijl de XHR Level 2 API ook inter-domein verkeer toelaat [65] . Websites die vertrouwen op de beperkingen van de oude XHR API, lopen nu gevaar voor kwetsbaarheden indien ze geen rekening houden met deze uitbreiding. Een voorbeeld van een bestaande website die op deze manier kwetsbaarheden opliep, is de mobiele versie van Facebook [1, 11]. Deze website nam een bepaalde webpagina als een parameter en laadde deze pagina in een XHR object, dewelke op zijn beurt de webpagina in het Facebook domein laadde. Via de oude XHR API kon de website enkel pagina's uit hetzelfde domein inladen, maar met de nieuwe API kan het XHR object plots data van alle mogelijke domeinen in het Facebook domein laden. Dit resulteert in een aantal nieuwe kwetsbaarheden aangezien een kwaadaardige pagina, eens geladen in het Facebook domein, willekeurige code kan uitvoeren binnen dat domein. De enige vereiste voor deze kwaadaardige website is de gepaste CORS hoofding toe te voegen, die het delen van de pagina mogelijk maakt.

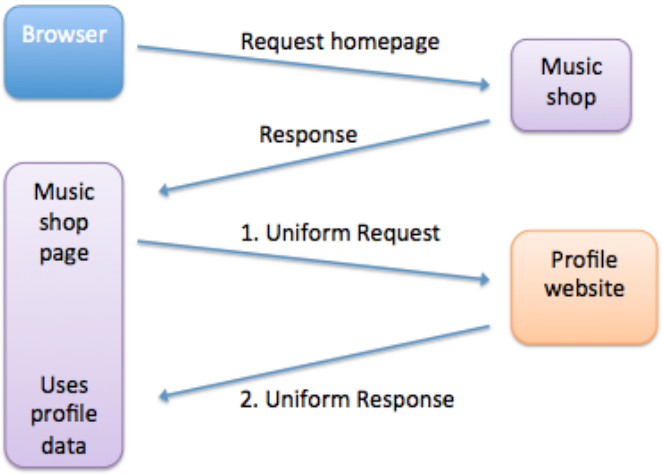
- *Doorverwijzingen*

  Een laatste probleem betreft het gebruik van doorverwijzingen, of *redirects*. Het werk van Bart et al. vermeldt verschillende problemen bij het gebruik van de *Origin header* [9]. Browsers hebben de gewoonte om bij doorverwijzingen de originele verzender te onthouden als origine. Stel bijvoorbeeld dat domein A een verzoek verstuurt naar domein B dat deze op zijn beurt doorverwijst naar domein A. In dit geval zou het toekomende verzoek op domein A `http://domeinA.com` als *Origin Header* bezitten. Indien domein B ter beschikking staat van de aanvaller kan dus elke pagina van domein A door domein B uitgevoerd worden in naam van domein A. Een mogelijke oplossing voor een dergelijk probleem zou kunnen bestaan uit het aanpassen van de *Origin Header* naar de meest recente *redirector*. Het toekomende verzoek zou in het vorige voorbeeld dan domein B als origine bezitten waarbij het probleem opgelost zou zijn.
  Bart et al. merken echter op dat zogenaamde open doorverwijzingen, of *Open Redirects*, hier alsnog voor problemen kunnen zorgen [41]. Open doorverwijzingen zijn pagina's die een bepaald adres als pagina meekrijgen en de gebruiker er vervolgens naar doorverwijzen (bv. `http://domeinA.com/redirect.php?page=domeinA.com/actie.html`). In dit geval kan de aanvaller domein B dus laten verwijzen naar de open doorverwijzing met als parameter een andere pagina van het domein A. Op die manier zal de voorgestelde oplossing niet meer werken. Een veilige oplossing zou eruit bestaan alle betrokken domeinen van een verzoek in de *Origin Header* te brengen. Op die manier kan de bestemming dus bepalen of een bepaald verzoek uitgevoerd mag worden door de ganse lijst van domeinen te bekijken. De *Origin Header* specificatie [13] heeft deze oplossing alvast overgenomen maar uit tests is gebleken dat browsers dit nog niet ondersteunen.
  Een specifiek CORS probleem dat betrekking heeft op de werking van open doorverwijzingen komt voor bij de verzending van een voorafgaand CORS verzoek. Wanneer een voorafgaand verzoek naar een goedwillig domein doorverwezen wordt naar een kwaadaardig domein, zou deze laatste de CORS hoofding teruggeven die betrekking heeft het goedwillig domein. Om deze manier kan domein B bijvoorbeeld toelating geven om bepaalde gegevens van domein A te verwijderen via de `DELETE` methode. Om deze kwetsbaarheid tegen te gaan, verbiedt de laatste versie van de CORS specificatie doorverwijzingen van voorafgaande verzoeken.

3.2.2 Uniform Messaging Policy

Aanpak

In tegenstelling tot CORS, bouwt het *Uniform Messaging Policy* [64] niet voort op de bestaande SOP en sessie mechanismen. UMP start met een propere lei waarbij interdomein verkeer nooit mag beschikken over cookies, HTTP authenticatie informatie of een *Origin Header* [58]. UMP stelt geen expliciet authenticatie mechanisme



FIGUUR 3.7: Een voorbeeldscenario dat gebruik maakt van UMP.

voor, maar raadt websites wel aan om een reeds goed onderzocht toegangscontrole mechanisme te implementeren bovenop het UMP mechanisme (bijvoorbeeld OAuth [39]).

UMP definieert twee type berichten, een *Uniform Request* en een *Uniform Response*. Een client verzendt een *Uniform Request* naar de server dat nooit impliciete authenticatie informatie bevat. Een website moet de eventueel vereiste authenticatie informatie expliciet toevoegen aan het verzoek via een parameter. De *Uniform Response* bevat het resultaat van het verzoek en één optionele hoofding die inter-domein data deling kan toelaten.

Voorbeeld scenario

Figuur 3.7 toont een schematisch overzicht van het inter-domein scenario waarin gebruik gemaakt wordt van het UMP ontwerp. De muziekwinkel zendt een *Uniform Request* naar de profiel website, inclusief expliciet aanwezige authenticatie informatie. Deze authenticatie gebeurt via OAuth [39], de exacte implementatie van dit mechanisme valt buien het bereik van deze tekst. De *Uniform Request* bevat geen *Origin Header*, cookies of HTTP authenticatie header.

De server controleert de authenticatie informatie en antwoordt met een *Uniform Response* waarin volgende header aanwezig is:

- **Access-Control-Allow-Origin:** Deze header bevat de waarde '*', welke aangeeft dat het antwoord met de website gedeeld mag worden.

Tot slot gebruikt de muziekwinkel de verkregen profielinformatie om de inhoud van de website te personaliseren.

Browser ondersteuning

UMP is tot op heden nog niet geïmplementeerd door browsers, ondanks de verhoogde aandacht die het gekregen heeft sinds de ontdekking van de CORS problemen beschreven [58]. Het lijkt erop dat UMP geïmplementeerd zal worden als een onderdeel van CORS [30] waarbij dezelfde XHR API gebruikt wordt. UMP kan gedeeltelijk gesimuleerd worden door CORS door de *credentials flag* af te zetten, waardoor enkel de *Origin Header* nog verwijderd moet worden.

Problemen

Indien de ontwikkelaar het toegangscontrole mechanisme op een correcte en veilige manier implementeert, verwijdert UMP de meeste problemen die inherent zijn aan de hiervoor beschreven CORS problemen, waaronder vooral de bedreiging van *Cross-Site Request Forgery*. CSRF aanvallen zijn enkel mogelijk omdat browsers sessie informatie impliciet mee verzenden met inter-domein verzoeken. Via het UMP mechanisme moet dit expliciet gebeuren waardoor dit soort aanvallen niet langer mogelijk is.

De verantwoordelijkheid van de beveiliging verplaatst dus van de client naar de server zijde, waardoor deze laatste verplicht is een gesofisticeerd authenticatie mechanisme als *Open Authentication* [39] te implementeren. De meeste hedendaagse websites maken gebruik van cookies en het lijkt een werk van lange adem vooraleer elke website zich zou hebben aangepast aan een veiligere variant.

3.3 Conclusie

Dit hoofdstuk besprak vooreerst de traditionele methodes om inter-domein communicatie te verwezenlijken. De nood aan dit soort communicatie heeft de ontwikkeling van deze methodes sterk in de hand gewerkt. Omdat veiligheid een knelpunt bleef, zijn recentelijk enkele nieuwe specificaties ontwikkeld, CORS en UMP. Momenteel ondersteunen browsers enkele CORS, maar dit hoofdstuk wees erop dat dit systeem te kampen heeft met veiligheidsproblemen. UMP lost de meeste problemen op, maar kan nog niet rekenen op de steun van de grote browser ontwikkelaars. Daarenboven verschuift UMP de verantwoordelijkheid meer naar de server-zijde, door de browser mechanismen die sessie informatie ondersteunen, niet toe te laten in inter-domein verkeer.

Hoofdstuk 4

Probleemstelling en aanpak

Dit hoofdstuk behandelt in sectie 4.1 de probleemstelling van deze masterproef die voortvloeit uit het onderzoek naar *Cross-Site Request Forgery* en veilige inter-domein communicatie, gevoerd in respectievelijk hoofdstuk 2 en 3. Sectie 4.2 beschrijft vervolgens hoe deze masterproef dit probleem aanpakt en formuleert een oplossing die de vereisten van een verbeterde CSRF tegenmaatregel omzet in een architectuur die rekening houdt met de meest recente ontwikkelingen op gebied van inter-domein communicatie.

4.1 Probleemstelling

4.1.1 Gebrek aan mobiele bescherming tegen CSRF

Er is een duidelijk gebrek aan bestaande CSRF maatregelen voor mobiele apparaten ondanks de voorspelde groei van mobiele apparaten met internettoegang. Enkel RequestPolicy en NoScript bieden een mobiele versie aan voor Mozilla Fennec, de mobiele variant van Mozilla Firefox. Voor andere mobiele gebruikers is helemaal geen CSRF bescherming voorhanden, ondanks de stijgende groep van onder andere Android gebruikers.

4.1.2 Gebrek aan ondersteuning van CORS door CSRF maatregelen

Een belangrijk probleem van alle besproken CSRF maatregelen is het gebrek aan ondersteuning van de in hoofdstuk 3 besproken inter-domein communicatiemechanismen. Terwijl het doel van deze technieken bestaat uit het mogelijk maken van veilige inter-domein communicatie, blijken geen van de besproken CSRF maatregelen in hoofdstuk 2 deze technieken expliciet te ondersteunen.

Cross-Origin Resource Sharing, aanvaard en gespecificeerd door de W3C groep[59], is het mechanisme dat browsers momenteel het best ondersteunen. Het onderzoek naar CORS wees erop dat deze techniek CSRF aanvallen niet in elk scenario kan vermijden. Een ondersteuning van dit mechanisme door een CSRF bescherming zal dus met zorg moeten gebeuren door rekening te houden met de gevonden complicaties.

4.2 Aanpak

Deze sectie beschrijft een modeloplossing die de probleemstelling moet aanpakken. De oplossing vertrekt hiervoor vanuit de systeemvereisten die subsectie 4.2.1 samenvat en die subsectie 4.2.2 tot slot omzet in een systeemarchitectuur.

4.2.1 Systeemvereisten

De basis voor deze vereisten volgt uit het gevoerde onderzoek naar bestaande maatregelen in sectie 2.3.2. Daaruit bleek CsFire, een extensie ontwikkeld voor Mozilla Firefox, het beste evenwicht te bieden tussen een optimale bescherming tegen *Cross-Site Request Forgery* en een ongeschonden gebruikerservaring tijdens het surfen. De eerste drie vereisten in deze sectie zijn dan ook grotendeels overgenomen van het onderzoek dat tot CsFire geleid heeft.

De vierde vereiste brengt het tweede deel van de probleemstelling in rekening door enerzijds de CORS functionaliteit toe te laten en anderzijds de werking van CORS aan te passen om een verbeterde bescherming tegen CSRF aanvallen te realiseren.

Onafhankelijk van gebruikersinvoer

Een eerste vereiste houdt in dat een CSRF bescherming niet mag rekenen op de mening van de gebruiker om een beslissing toe te laten.

Dit omdat de extensie niet mag veronderstellen dat elke gebruiker zich gewaar wordt van de gevaren die bepaalde verzoeken inhouden. Zelfs de meest ervaren gebruiker kan niet altijd op de hoogte zijn van de externe websites die een bepaalde website nodig heeft om correct te functioneren.

Ten tweede is het ook niet praktisch haalbaar om de gebruiker telkens te storen bij elk inter-domein verzoek. De inleiding benadrukte namelijk het stijgend aantal websites dat nood heeft aan inter-domein verkeer.

Compatibel met web 2.0 technologieën

Een tweede vereiste betreft de ongeschonden bruikbaarheid van bestaande web 2.0 technologieën. De standaard werking van CsFire blokkeert geen enkel verzoek, wat leidt tot een betere bruikbaarheid vergeleken met de overige extensies besproken in sectie 2.3.2.

Veilige standaard die verfijning toelaat

De standaard instellingen zouden de meest veilige situatie moeten garanderen bij algemeen gebruik van het internet. De hier ontwikkelde tegenmaatregel bouwt verder op het CsFire beleid dat werd ontwikkeld na een extensieve evaluatie van dataverkeer bij 15 proefpersonen gedurende twee weken.

Desondanks zal de standaard werking niet optimaal zijn voor elke situatie. Veel websites maken bijvoorbeeld gebruik van de Facebook API om bepaalde delen van een webpagina te personaliseren. Via het standaard beleid worden deze verzoeken

naar Facebook telkens gestript van authenticatie gegevens, wat de werking van dit systeem blokkeert. De toelating van uitzonderingen, op een veilige manier, kan zorgen voor een verbeterde gebruikservaring.

Ondersteuning van het CORS mechanisme

Aangezien CORS specifiek ontwikkeld werd om veilige inter-domein communicatie mogelijk te maken, is een gepaste ondersteuning door de hier ontwikkelde extensie onontbeerlijk. Dit houdt in dat de extensie rekening moet houden met de volgende gebreken van CORS, die voortkwamen uit het onderzoek gevoerd in hoofdstuk 3:

- Ten eerste beschermt CORS enkel verzoeken die via de CORS API uitgevoerd worden. *Cross-Site Request Forgery* is nog steeds mogelijk via de oude manieren, eigen aan de werking van het SOP mechanisme. Zo moeten bijvoorbeeld afbeeldingen die intern-domein geladen worden, nog steeds gestript worden van eventuele authenticatie informatie.
- Ten tweede zijn ook eenvoudige CORS verzoeken niet beschermd tegen CSRF, ondanks dat ze door de CORS API uitgevoerd worden. Deze zijn enkel veilig wanneer de server die het CORS verzoek ontvangt, hier correct mee omgaat. De correcte behandeling van CORS verzoeken is echter niet verzekerd, aangezien een maatregel niet kan veronderstellen dat elke website reeds ondersteuning biedt voor dit recent ontwikkelde mechanisme.

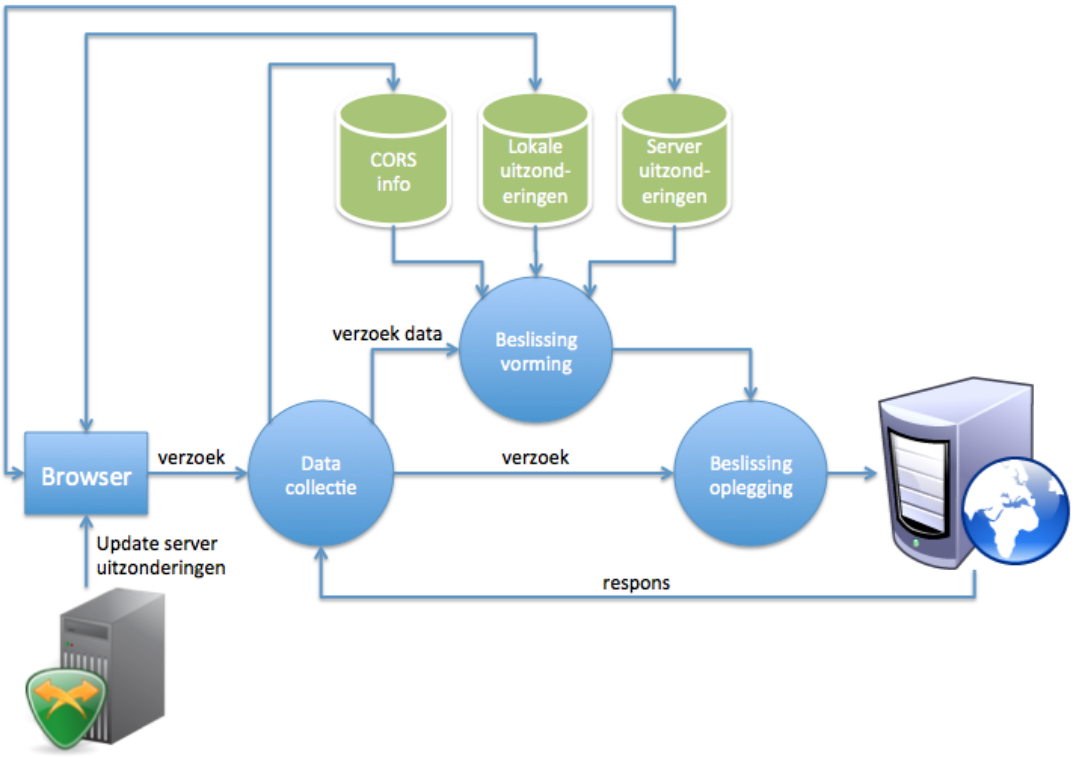
4.2.2 Systeemarchitectuur

Deze sectie licht enerzijds de algemene architectuur van de ontwikkelde tegenmaatregel toe en gaat anderzijds dieper in op elk onderdeel. Figuur 4.1 geeft de algemene architectuur schematisch weer. Ze bestaat uit drie grote onderdelen: een datacollectie, een beslissingsvorming en een beslissingsoplegging.

Datacollectie

Vooraleer de extensie een beslissing kan nemen, dient ze de volgende informatie te verzamelen:

- **Informatie over het uitgaande verzoek**
  - De *oorsprong* van het verzoek. Het schema (bv. `http`), de domeinnaam (bv. `example`), de domein extensie (bv. `com`), de poort (bv. `80`) en het pad naar de pagina (bv. `/een/pad/naar/de/pagina.html`) moeten gekend zijn. De oorsprong kan leeg zijn wanneer een gebruiker bijvoorbeeld een adres in de adresbalk invoert of een bladwijzer opent.
  - De *bestemming* van het verzoek. De elementen die hier gekend moeten zijn, komen overeen met de benodigde elementen van de oorsprong.
  - De gebruikte *methode* in het verzoek. Bv. de `GET` methode.



Figuur 4.1: De algemene architectuur van de tegenmaatregel.

– De aanwezigheid van *parameters* moet gekend zijn. Bij een `GET` verzoek bevinden parameters zich in de URL terwijl een `POST` verzoek deze ook in het lichaam van het verzoek kan plaatsen.
– Verder moet geweten zijn of een verzoek het gevolg is van *gebruikersinteractie* op een webpagina. Gebruikersinteractie wordt in dit geval beperkt tot een klik op een `<a>` element of de invoer knop van een `<form>` element.
– De extensie moet nagaan of het om een *CORS verzoek* gaat. Daarenboven moet ook geweten zijn of het om een eenvoudig of geavanceerd verzoek gaat.
– Er moet geweten zijn of een verzoek het gevolg is van een *doorverwijzing*. De dataverzameling maakt op dit punt een onderscheid tussen twee soorten doorverwijzingen.

  Ten eerste zijn er de gewone doorverwijzingen die geen gevolg zijn van de CORS API. De meeste browsers behouden bij een opéénvolging van verwijzingen de origine van het eerste verzoek als origine doorheen de ganse reeks van doorverwijzingen, ondanks eventuele tussenliggende interdomein verzoeken. Zoals sectie 3.2.1 reeds aanhaalde, houdt dit echter een *confused deputy* gevaar in. Dit beslissingsalgoritme hanteert dezelfde

Veldnaam	**Beschrijving**
ID:	Het unieke identificatienummer van deze CORS informatie.
DEST:	De webpagina waarvan de CORS informatie afkomstig is.
ACAO:	De inhoud van de `Access-Control-Alow-Orgin` hoofding met de toegelaten origine of de `"*"` waarde.
METHOD:	De methode waarvoor deze regel geldt.
HEADER:	De aangepaste hoofding waarvoor deze regel geldt (indien leeg geldt ze voor geavanceerde verzoeken zonder aangepaste hoofding).
ACMA:	De inhoud van de `Access-Control-Max-Age` hoofding met de maximale geldigheid van de CORS informatie.
ACAC:	De inhoud van de `Access-Control-Alow-Credentials` hoofding die aangeeft of het verzoek sessie informatie mag gebruiken.
TIMESTAMP:	Het tijdstip waarop de CORS informatie werd verkregen.

Tabel 4.1: De structuur van de CORS database.

oplossing als CsFire, door alsnog de origine van de laatste verwijzing te gebruiken bij de behandeling van een inter-domein verzoek. Deze oplossing neemt spijtig genoeg niet alle problemen weg. Sectie 3.2.1 wees namelijk reeds op de blijvende gevaren van open doorverwijzingen, of *Open Redirects*.

Ten tweede zijn er de doorverwijzingen na een CORS verzoek. Aangezien de ondersteuning van CORS doorverwijzingen verschilt tussen browsers, zal de ontwikkelde extensie rekening moeten houden met de werking van de specifieke browser.

– Tot slot moet het gebruik en de inhoud van *cookies* en het gebruik van de *HTTP authenticatie* methode gekend zijn.

- **Informatie van de inkomende *respons***

  – Wanneer de extensie een antwoord ontvangt op een voorafgaand CORS verzoek, dient deze de informatie op te slagen in een daarvoor bestemde CORS database. Tabel 4.1 geeft de vereiste velden weer van deze database. Het antwoord op een voorafgaand CORS verzoek kan afhankelijk zijn van de origine van het verzoek en de aangevraagde methode. Daarom houdt de database een aparte rij bij voor elk `DEST/ACAO/METHOD` koppel.

  – Indien de respons een doorverwijzing aangeeft via de *Location Header*, moet de datacollectie dit onthouden.

- **Lokale uitzonderingen**

  Ook het manueel toevoegen van lokale uitzonderingen behoort tot de datacollectie. Een gebruiker kan via de browser uitzonderingen toevoegen, aanpassen en verwijderen. Een regel bevat de eigenschappen van de origine, de eigenschappen van de bestemming, de kenmerken van het verzoek en de uit te voeren beslissing.

- **Server uitzonderingen**

  Een laatste element van de datacollectie betreft de automatische toevoeging van uitzonderingen die van een specifieke server afkomstig zijn. Deze komen eveneens in een database terecht, zodat de beslissingsvorming ze later kan controleren bij het maken van een beslissing.

Vormen van een beslissing

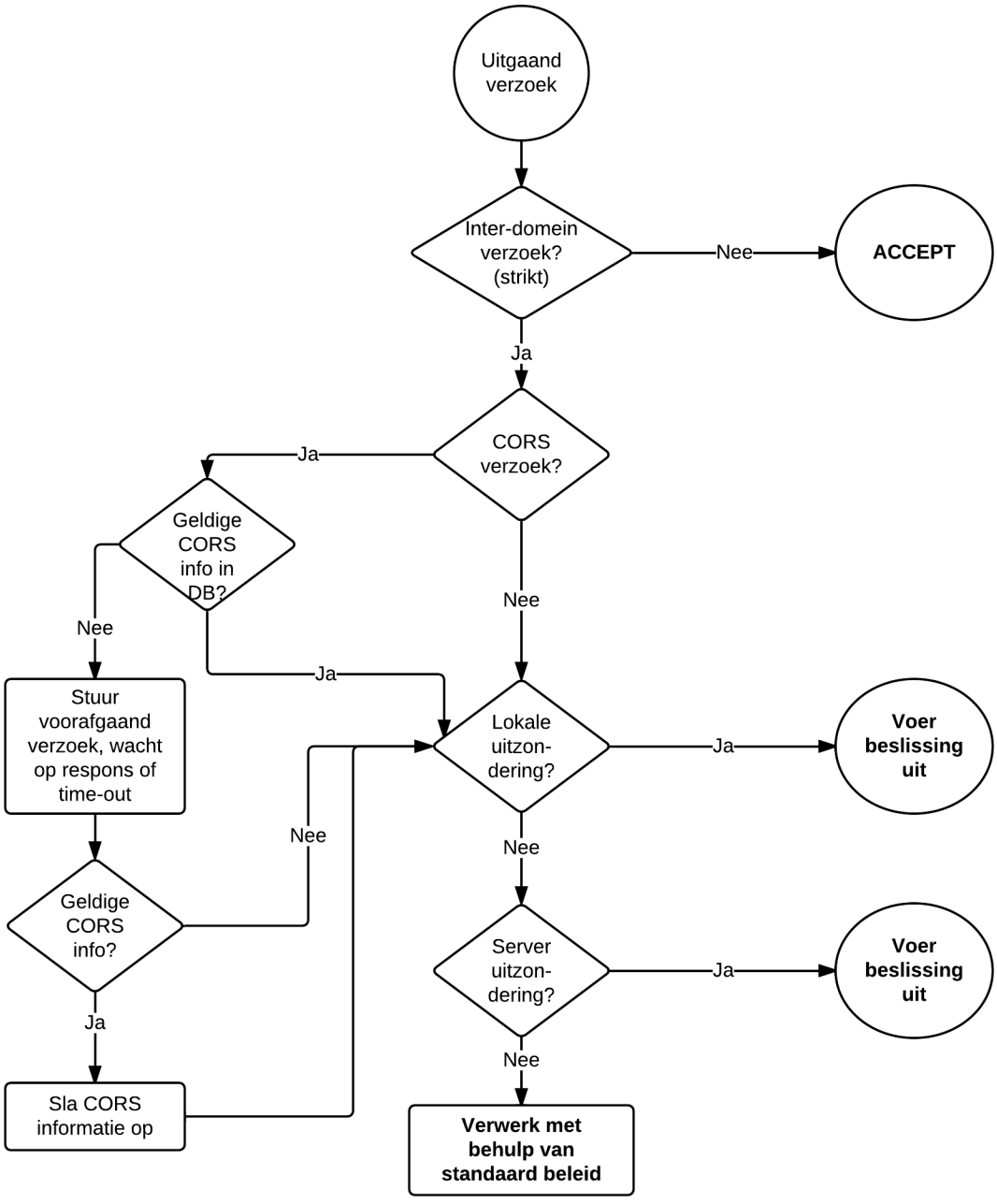
Dit onderdeel van de architectuur bevat het beslissingsalgoritme dat gebruik maakt van de verzamelde data. Het algoritme kan leiden tot volgende beslissingen:

- **ACCEPT:** Het algoritme accepteert het verzoek.
- **BLOCK:** Het algoritme blokkeert het verzoek.
- **ASK:** Het algoritme vraagt aan de gebruiker wat er met het inter-domein verzoek moet gebeuren.
- **STRIP (integraal):** Het algoritme verwijdert de eventueel aanwezige sessie informatie. Het verwijdert zowel de aanwezige cookies als de gebruikte HTTP authenticatie informatie uit het verzoek. Deze beslissing kan bepaalde uitzonderingen maken om sommige cookie variabelen alsnog toe te laten.
- **STRIP(cookie):** Het algoritme verwijdert de eventueel aanwezige cookies. Deze beslissing kan eveneens bepaalde uitzonderingen maken om sommige cookie variabelen alsnog toe te laten.
- **STRIP(AH):** Het algoritme verwijdert de eventueel aanwezige HTTP authenticatie hoofding uit het uitgaande verzoek.

Figuur 4.2 geeft het beslissingsalgoritme weer om tot één van de voorgaande beslissingen te komen. De volgende lijst overloopt de verschillende stappen van het algoritme:

1. **Inter-domein verzoek**

   Een eerste stap gaat na of het om een inter-domein verzoek gaat. Het beleid maakt gebruik van de strikte definitie van origine en houdt dus rekening met het schema, de domeinnaam en de poort. Indien het verzoek volgens deze definitie inter-domein is, gaat het algoritme over naar de volgende stap . In het andere geval legt het de `ACCEPT` beslissing op.



FIGUUR 4.2: Een flowchart van het algemene beslissingsmodel.

2. **CORS verzoek**

   Een volgende stap gaat na of het verzoek afkomstig is van de CORS API. Indien dit het geval is, controleert het algoritme of er reeds geldige informatie beschikbaar is in de CORS database. Geldige informatie houdt in dat de gebruikte methode, aangepaste hoofding, origine en bestemming moeten overeenkomen. Dit zou bijvoorbeeld het geval kunnen zijn wanneer het om een geavanceerd CORS verzoek gaat. Bij een dergelijk verzoek verstuurt de browser namelijk een voorafgaand verzoek naar de betreffende pagina. Wanneer een respons zou binnengekomen zijn met geldige CORS informatie, werd deze door de datacollectie opgeslagen in de CORS database.

   Indien er nog geen CORS informatie over de specifieke bestemming van het verzoek aanwezig is, verzendt het algoritme zelf een voorafgaand verzoek. Op deze manier dwingt het algoritme ook voor eenvoudige verzoeken een voorafgaand verzoek af. Het beslissingsproces gebruikt deze verkregen informatie dan later om een verzoek al dan niet toe te laten.

   Een laatste opmerking in deze fase betreft de geldigheid van de aanwezige informatie. De opgeslagen CORS informatie heeft namelijk een beperkte duur, afhankelijk van de waarde van de `Access-Control-Max-Age` hoofding. Indien deze maximale tijdslimiet overschreden wordt, is de informatie niet langer geldig en moet de browser een nieuw voorafgaand verzoek naar de server verzenden.

3. **Lokale uitzonderingen**

   Een volgende stap kijkt na of de gebruiker zelf een uitzondering heeft ingevoerd die betrekking heeft op het behandelde verzoek. Indien het algoritme een overeenkomst vindt, voert het de bijhorende beslissing uit. Zo niet, gaat het over naar de volgende stap.

4. **Server uitzonderingen**

   Deze stap toetst de eigenschappen van het verzoek aan de eigenschappen van de server uitzonderingen. Indien het algoritme een overeenkomst vindt, voert het de bijhorende beslissing uit. Zo niet, gaat het over naar de volgende stap.

5. **Standaard beleid**

   Deze laatste stap behandelt het standaard beleid dat op autonome wijze beslist wat er met een verzoek moet gebeuren. De flowchart in figuur 4.3 geeft de stappen weer van dit onderdeel van het beslissingsalgoritme. Volgende lijst overloopt de verschillende stappen:

   - *Inter-domein verzoek*

     Indien het niet om een inter-domein verzoek gaat volgens de afgezwakte definitie, legt het algoritme de `ACCEPT` beslissing op. Zo niet wordt overgegaan naar de volgende stap.

- *Raadpleging CORS database*

  Indien het om een CORS verzoek gaat, raadpleegt het algoritme de CORS database om na te gaan of er geldige informatie beschikbaar is voor het betreffende verzoek. Ook hier betekent geldig dat zowel de methode, bestemming, origine als aangepaste hoofding moeten overeenkomen. Indien de bestemmende server CORS ondersteunt, zal de datacollectie het antwoord van het voorafgaand verzoek sowieso hebben opgeslagen in de CORS database (zowel voor eenvoudige als geavanceerde verzoeken).

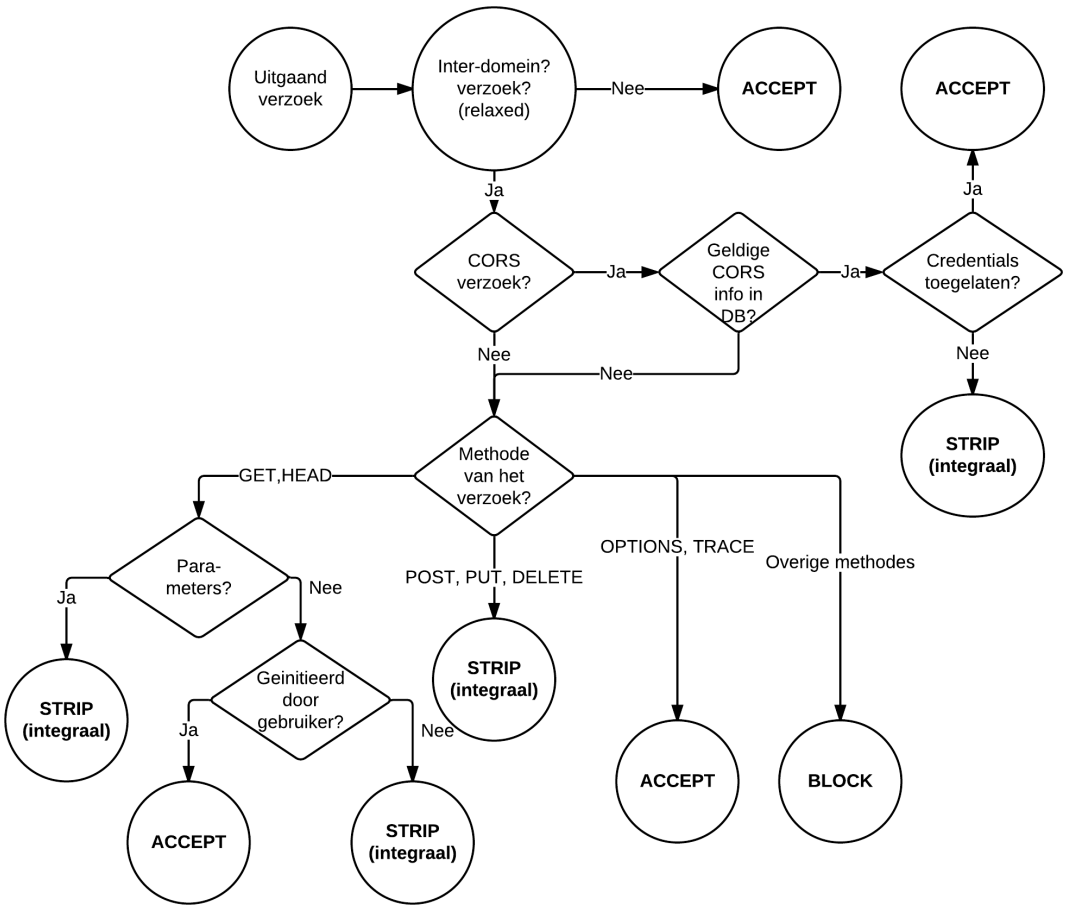
  Bij de aanwezigheid van CORS informatie, zal het algoritme de uiteindelijke beslissing maken op basis van het `Access-Control-Allow-Credentials` veld. Indien de waarde hiervan `true` is, legt het de `ACCEPT` beslissing op, in het andere geval de `STRIP`(integraal) beslissing.

- *Methode, parameters en gebruikersinteractie*

  Indien het niet om een CORS verzoek gaat, bepaalt het algoritme de beslissing op basis van de methode, de aanwezigheid van parameters en de eventuele gebruikersinteractie. Dit is bijna volledig analoog aan de aanpak die CsFire voorstelt om inter-domein verzoeken te behandelen.

  - *GET en HEAD*
    Het algoritme neemt voor de `GET` en `HEAD` methode de `STRIP`(integraal) beslissing, behalve als het gaat om een verzoek zonder parameters, geïnitieerd door de gebruiker. Dan past het algoritme de `ACCEPT` beslissing toe.
  - *POST, PUT en DELETE*
    Voor verzoeken die gebruik maken van de `POST`, `PUT` of `DELETE` methode, legt het algoritme de `STRIP`(integraal) beslissing op. Voor `POST` verzoeken is deze keuze logisch omdat dit soort verzoeken ook via formulieren kan gebeuren. In principe zouden verzoeken met de `PUT` of `DELETE` methode enkel kunnen gebeuren via een geavanceerd CORS verzoek. Dit is het geval sinds de aanpassing van de HTML5 specificatie die dateert van juni 2010 [63], die vermeldt dat formulieren niet langer de mogelijkheid bezitten om `PUT` en `DELETE` verzoeken te versturen. Een `STRIP` beslissing is hier dus niet meer noodzakelijk omdat de browser zelf de correcte beslissing afdwingt door het verzoek te blokkeren. Om eventuele toekomstige problemen te vermijden bij nieuwe aanpassingen aan de specificatie, past het algoritme hier echter alsnog de `STRIP`(integraal) beslissing toe.
  - *OPTIONS en TRACE*
    Deze methodes krijgen de beslissing `ACCEPT` opgelegd omdat ze volgens de specificatie geen randeffecten mogen opleveren [60]. Bovendien haalde sectie 3.2.1 reeds aan dat een voorafgaand verzoek, die gebruik maakt van de `OPTIONS` methode, nooit sessie informatie mag bevatten.
  - *Overige methodes*

Het algoritme blokkeert verzoeken die gebruiken maken van een andere methode dan hiervoor beschreven. Indien browsers nieuwe methodes zouden invoeren, moeten deze eerst geëvalueerd worden vooraleer het standaard beleid een beslissing kan bevatten.



Figuur 4.3: Een flowchart van het standaard beleid.

Uitvoeren van de beslissing

Het laatste onderdeel van de systeemarchitectuur in figuur 4.1 verzorgt de correcte uitvoering van de beslissing. het algoritme moet een verzoek kunnen blokkeren, cookies selectief kunnen verwijderen en HTTP authenticatie info kunnen verwijderen.

4.3 Conclusie

De beschreven aanpak biedt een goede balans tussen een doeltreffende CSRF bescherming en een ongeschonden bruikbaarheid, door te vertrekken vanuit de basiswerking van CsFire. De nieuwigheid zit hem enerzijds in de ondersteuning van CORS en anderzijds in het gebruik van CORS om een dynamisch beleid te kunnen realiseren. Terwijl CsFire enkel kan rekenen op manuele uitzonderingen (lokaal of gedownload van een vooraf vastgelegde server), maakt de nieuwe extensie gebruik van de CORS ondersteuning op servers om uit de standaard werking van CsFire te stappen.

Om tegemoet te komen aan het eerste aspect van de probleemstelling, implementeert het volgend hoofdstuk het ontwikkeld beslissingsalgoritme op twee verschillende mobiele browsers.

Hoofdstuk 5

Implementatie van een tegenmaatregel voor mobiele browsers

Dit hoofdstuk zet de in hoofdstuk 4 ontwikkelde aanpak om in een implementatie voor twee verschillende mobiele browsers.

Ten eerste beschrijft sectie 5.1 de ontwikkeling van CsFennec, een extensie voor Mozilla Fennec die grotendeels gebaseerd is op de implementatie van CsFire. Bovendien verandert ze ook de originele implementatie van CsFire zodat ook deze CORS ondersteunt.

Ten tweede behandelt sectie 5.2 de ontwikkeling van CsDroid, een aangepaste versie van de Android Browser. Deze laatste implementatie kan als inspiratie dienen voor andere platformen die gebaseerd zijn op dezelfde architectuur.

5.1 Mozilla Fennec

Mozilla Fennec, de werknaam van de mobiele versie van Mozilla Firefox, is een browser die momenteel nog volop in ontwikkeling is. De ondersteuning door apparaten is momenteel dan ook beperkt, zoals aangegeven op de officiële website[1]. Het grote voordeel van deze browser is dat ze voortbouwt op dezelfde architectuur als Firefox, namelijk de open-source *Gecko engine*[2]. Hierdoor voorziet deze browser gelijkaardige extensie mogelijkheden. Het is dankzij deze mogelijkheden dat het eerste deel van de ontwikkeling van CsFennec, de naam van deze eerste extensie, zeer vlot verliep. Dit omdat de implementatie heel wat code van de CsFire extensie kan hergebruiken. Het tweede deel voegt aan deze basisbescherming de CORS ondersteuning toe zoals in hoofdstuk 4 beschreven.

[1] http://wiki.mozilla.org/Fennec
[2] http://wiki.mozilla.org/Gecko:Home_Page

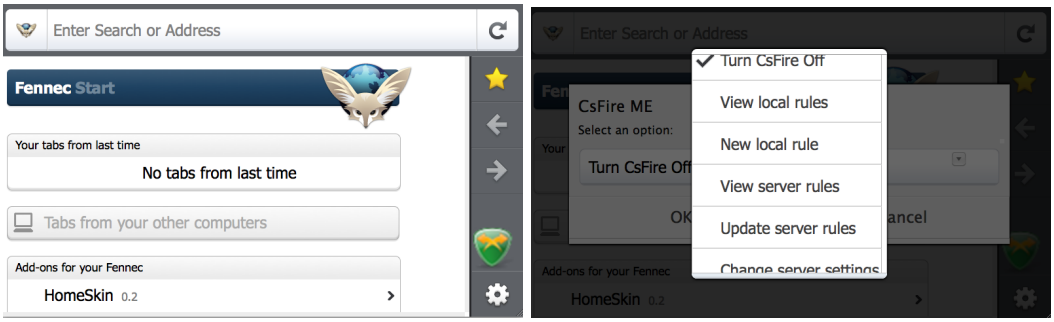
5.1.1 Basisbescherming voor CSRF

Een Firefox of Fennec extensie bestaat in principe uit twee aparte blokken, een XPCOM en een XUL onderdeel. XPCOM, wat staat voor *Cross-Platform Component Object Model*, is een platform onafhankelijk raamwerk dat toelaat componenten te ontwikkelen met een specifieke functionaliteit. XUL bestanden, waarbij XUL staat voor *XML-based User-interface Language*, bevatten daarentegen code voor de gebruiksinterface die verschillend is tussen de verschillende Mozilla applicaties.

Ook CsFire maakt gebruik van dit onderscheid waardoor CsFennec de XPCOM elementen integraal kan overnemen. De implementatie moet enkel de elementen met betrekking tot de gebruikers in- en output aanpassen. Omdat dit hoofdstuk de nadruk legt op de eigen contributie, bespreekt het enkel de belangrijkste verschillen met CsFire. De overige details van de implementatie kan de lezer in het onderzoek rond CsFire terugvinden [48].

Gebruikersinterface

De gebruiksinterface van een mobiele applicatie verschilt van een desktop applicatie omwille van de kleinere resolutie. In dit geval voorziet de implementatie dus niet enkel aangepaste code, maar ook een nieuwe bedachte interface. CsFennec maakt gebruik van een XUL element waarbij, vergelijkbaar met de desktop variant, een statussymbool aanwezig is in het rechtse deel van de gebruiksinterface. Verder voorziet de extensie ook een nieuw menu dat hoofdzakelijk gebruik maakt van *drop-down* lijsten maar waarin alle functionaliteit van de originele CsFire extensie vervat zit. Figuur 5.1 toont enkele beelden van de nieuwe gebruiksinterface. Tot slot is ook de afhandeling van de `ASK` beslissing aangepast aan het kleinere scherm.



Figuur 5.1: Enkele screenshots van CsFennec.

Opvangen gebruikersinteractie

De originele code van CsFire voor het opvangen van een klik was grotendeels geïnspireerd op de code van RequestPolicy[3]. Ondanks dat RequestPolicy ook een mobiele versie heeft met aangepaste Fennec code voor het opvangen van een klik,

[3]RequestPolicy werd besproken in hoofdstuk 2 bij de sectie over bestaande maatregelen.

bleek deze code na tests op de laatste beta versies van Mozilla Fennec niet te werken. De reden hiervoor lag in een belangrijk verschil tussen Firefox en Fennec. Terwijl Firefox toegang tot de webinhoud toelaat binnen het XUL proces, verplicht Fennec het gebruik van zogenaamde *Content Process Event Handlers*[4]. Firefox heeft de mogelijkheid om binnen het XUL proces een klik op te vangen en vervolgens het aangeklikte DOM element op te vragen. In Fennec beschikt elk XUL proces daarentegen over een *messageManager* object, dat een implementatie verzorgt van het *nsIChromeFrameMessageManager* element. Via dit element kan een extensie verschillende *messageListeners* toevoegen, die dan de opvang van een klik kunnen behandelen. Volgende code illustreert het toevoegen van een *listener* in het XUL proces:

```
messageManager.loadFrameScript("resource://ui_handler.js", true);
messageManager.addMessageListener("linkclick",
function(m) {
    var target = m.json.target;
    var origin = m.json.origin;
    registerUserInteraction(target, origin);
}
```

De code van de listener `ui_handler.js` kan er dan als volg uit zien:

```
addEventListener("click",
    function(e) {
        if (e.target instanceof
                  Components.interfaces.nsIDOMHTMLAnchorElement)
        {
           sendSyncMessage("linkclick",
               {
                 target : e.target.href,
                 origin : e.target.ownerDocument.URL
               }
           );
        }
    },
false);
```

In dit geval behandelt de *listener* de opvang van een klik en verstuurt ze de boodschap met de benodigde informatie naar de functie die gelinkt is aan de naam `linkclick`.

[4] http://wiki.mozilla.org/Content_Process_Event_Handlers

5.1.2 Ondersteuning van CORS

Deze sectie gaat in op de noodzakelijke aanpassingen om de ondersteuning van CORS te verzekeren. Zoals de aanpak aangaf, moet CsFennec een CORS database aanmaken en het aangepast beslissingsmodel implementeren.

CORS database

Het aanmaken en beheren van persistente data binnen het Mozilla platform kan met behulp van een SQLite database[5]. Een extensie kan zelf een database met tabellen aanmaken en ze vervolgens met data vullen. CsFire had op deze manier reeds een tabel aangemaakt voor het bewaren van lokale en server uitzonderingen. Op gelijkaardige wijze maakt CsFennec nu ook een SQLite tabel aan met de velden zoals in tabel 4.1 beschreven.

CORS verzoeken onderscheppen

Het Mozilla platform implementeert de CORS API via de XMLHttpRequest (XHR) Level 2 API [38]. Het beslissingsmodel moet weten of een verzoek al dan niet afkomstig is van een XHR object. CsFennec registreert elk verzoek dat de browser verzendt waardoor het beslissingsproces dit nadien kan behandelen. Bij deze registratie is het `httpChannel` object beschikbaar met daarin alle informatie over het verzoek. Het is via dit object dat CsFennec te weten komt of het om een CORS verzoek gaat. De extensie kan namelijk de `notificationCallbacks` eigenschap opvragen van het `httpchannel` object en dan controleren of deze eigenschap bestaat. Indien het antwoord positief is, gaat het om een CORS verzoek.

```
try {
    var callbacks = httpChannel.notificationCallbacks;
    if (callbacks) {
        XHR = true;
    }
} catch (exc) {
    if (exc.name == "NS_NOINTERFACE") {
        XHR = false;
    }
}
```

Indien uit voorgaande code blijkt dat het om een XHR verzoek gaat, geeft de datacollectie dit door aan de beslissingsvorming, zodat het standaard beleid hier rekening mee kan houden.

Opvangen van antwoorden op voorafgaande verzoeken

CsFire registreert reeds binnenkomende antwoorden voor de goede behandeling van doorverwijzingen en het correct verwijderen van de HTTP authenticatie hoofding.

[5] http://www.sqlite.org/

Op diezelfde plaats vangt CsFennec nu ook de antwoorden op van voorafgaande verzoeken. Bij het binnenkomen van een antwoord dat gebruik maakt van de `OPTIONS` methode, leest de datacollectie de informatie in die zich in de hoofding bevindt. Indien er geldige CORS informatie aanwezig is, slaat de datacollectie deze informatie op in de CORS database.

Uitvoeren van voorafgaande verzoeken

Zoals hoofdstuk 4 aangaf, zijn eenvoudige CORS verzoeken niet impliciet beschermd tegen CSRF. Daarom zal de hier ontwikkelde extensie ook bij dit soort verzoeken een voorafgaand verzoek verzenden.

Indien een CORS verzoek langs de beslissingsvorming passeert, gaat de extensie na of er informatie in de CORS database aanwezig is. Indien dit niet het geval is of de geldigheidsduur van de eventuele CORS informatie vervallen is, verzendt het beslissingsalgoritme een voorafgaand verzoek. Dit zal enkel gebeuren wanneer geen geldige CORS informatie aanwezig is, wat enkel het geval kan zijn bij eenvoudige CORS verzoeken. Voor geavanceerde CORS verzoeken moet de browser volgens de specificatie reeds een voorafgaand verzoek verzonden hebben, zodat de datacollectie reeds CORS informatie heeft ontvangen en opgeslagen.

Het beslissingsmodel wacht tot de ontvangst van het antwoord op het voorafgaand verzoek of tot een time-out is opgetreden. De verzending van een voorafgaand verzoek gebeurt via een asynchroon XHR verzoek, zoals in volgend voorbeeld:

```
if(XHR) {
  var xhr =
    Components.classes["@mozilla.org/xmlextras/xmlhttprequest;1"]
              .createInstance(Components.interfaces.nsIXMLHttpRequest);
  xhr.open("OPTIONS", url, false);
  xhr.send(null);
}
```

CORS doorverwijzingen

Het onderzoek naar CORS wees reeds op de bestaande problemen met doorverwijzingen. Momenteel hebben Mozilla Firefox en Fennec de meest recente specificatie van de *Origin Header* nog niet geïmplementeerd. Ze hanteren daarentegen volgende twee regels[28]:

- Indien het eerste verzoek in hetzelfde domein gebeurt, mag er één keer inter-domein doorverwezen worden. De browser voegt de *Origin Header* op dat moment toe.
- Van het moment dat een verzoek of een doorverwijzing inter-domein gebeurt, zijn doorverwijzingen niet langer toegestaan.

CORS doorverwijzingen zijn met andere woorden niet toegelaten binnen Firefox of Fennec waardoor een verdere aanpassing aan het standaard beleid niet vereist is.

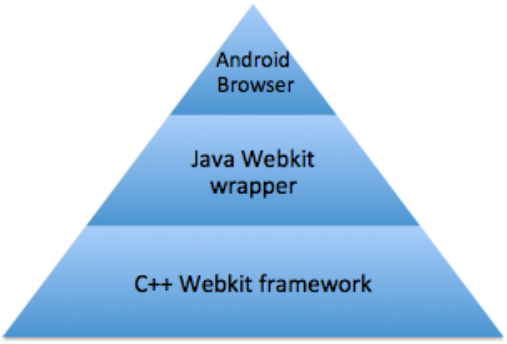
Standaard beleid

Een laatste aanpassing dient te gebeuren in het standaard beleid van CsFennec. Ten eerste gaat de extensie na of er geldige CORS informatie in de CORS database beschikbaar is. Indien dit het geval is, neemt het algoritme een beslissing op basis van de waarde van de het `Access-Control-Allow-Credentials` veld. Als dit veld de waarde `true` heeft, neemt het algoritme de `ACCEPT` beslissing, in elk ander geval de `STRIP` (integraal) beslissing.

5.2 Android Browser

De Android browser is standaard geïnstalleerd op alle versies van het Google Android systeem en maakt gebruik van de Webkit engine[10]. Het grote verschil met Mozilla Firefox is het gebrek aan een uitgebreid extensieplatform. Aanpassingen aan de Android Browser dienen dan ook in de broncode zelf te gebeuren waarbij een hercompilatie van het systeem vereist is.

Bij de ontwikkeling van de extensie werd gebruik gemaakt van de emulator en de *Android Debug Bridge*(ADB) die Android op zijn website beschikbaar stelt[6]. De emulator maakt het bezit van een Android apparaat overbodig door middel van een geëmuleerde versie van het Android besturingssysteem waarop de Android Browser geïnstalleerd staat. De ADB laat toe om specifieke debug informatie naar de terminal te schrijven, wat de ontwikkeling positief beïnvloedt. Verder werd voor de ontwikkeling gebruik gemaakt van Eclipse en XCode om respectievelijk Java en C++ code te compileren. De aanpassingen aan de C++ code bleven wel beperkt tot enkele debuglijnen om de onderliggende werking te doorgronden.



Figuur 5.2: De drie grote onderdelen waaruit de Android Browser is opgebouwd.

Figuur 5.2 geeft de algemene architectuur van de Android Browser weer die uit drie grote onderdelen bestaat. De onderste laag bevat een aangepaste versie van Webkit in C++ code waarin de kern van de functionaliteit verwerkt zit[7]. Hier is vervolgens een *Java Webkit Wrapper* rond gebouwd die de C++ code toegankelijk

[6] http://developer.android.com

[7] Android maakt gebruik van revisie 54731 van Webkit.

maakt binnen de Java code en Java versies bevat van de belangrijkste Webkit klassen. De Android Browser applicatie maakt tot slot via deze laag gebruik van de Webkit functionaliteit.

CsDroid, de naam van deze implementatie, is een aangepaste versie van de Android Browser waarbij de aanpassingen vooral in de *Java Webkit Wrapper* plaatsvinden. Enerzijds opdat een eventuele CSRF bescherming voor andere Webkit gebaseerde browsers zich zou kunnen inspireren op deze code. Anderzijds omdat de CSRF bescherming niet te ver van de applicatie code verwijderd is en toegang tot de standaard Android mogelijkheden relatief eenvoudig blijft. Uit ervaring bleek dat het Android systeem nog niet helemaal stabiel is waardoor soms onverklaarbare fouten of crashes voorkomen. Hierdoor bevat CsDroid niet altijd alle aspecten zoals hoofdstuk 4 voorschrijft, al blijft dit beperkt tot enkele kleine mankementen.

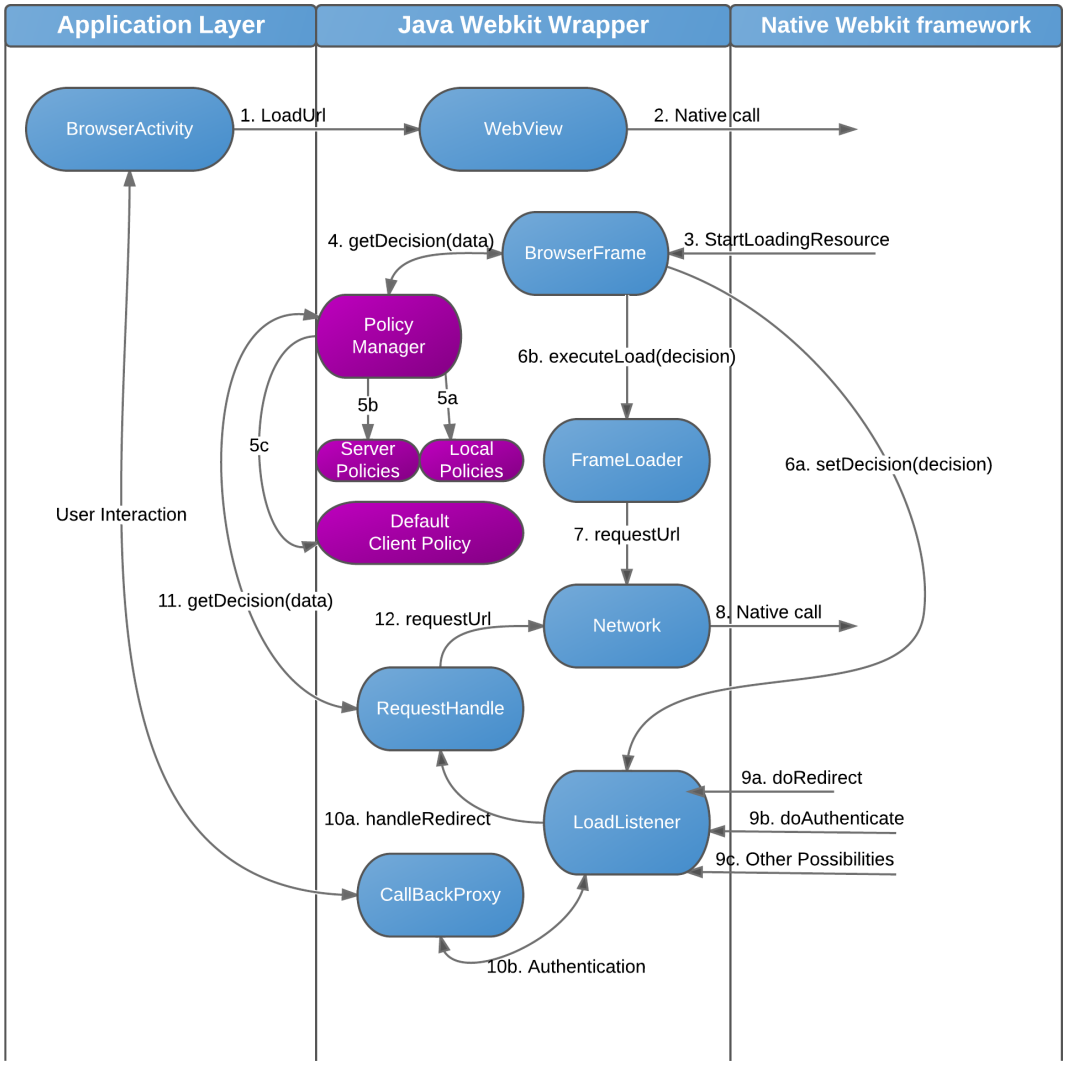
5.2.1 Basisbescherming voor CSRF

Dit eerste gedeelte behandelt de implementatie van de CSRF basisbescherming zoals CsFire deze voorstelde. De rest van deze sectie hanteert de onderverdeling van hoofdstuk 4 met eerst de dataverzameling, vervolgens de beslissingsvorming en tot slot de beslissingsoplegging. Figuur 5.3 ondersteunt de uitleg met een vereénvoudigde voorstelling van het laden van een webpagina door de Android Browser.

Datacollectie

Deze sectie vertelt waar de verschillende elementen van de datacollectie zich bevinden in figuur 5.3.

- De plaats van de data onderschepping verraadt of het al dan niet om een doorverwijzing gaat. Indien het om een doorverwijzing gaat, ontvangt het `RequestHandle` object het verzoek. Als het om een gewoon verzoek gaat, onderschept het `BrowserFrame` object het verzoek.
- De *oorsprong*, *bestemming* en *methode* van het uitgaand verzoek worden verkregen in het `BrowserFrame` object na stap 3 of in het geval van een doorverwijzing, in het `RequestHandle` object na stap 10a.
- Na stap 3 of 10a is ook de aanwezigheid en inhoud van eventuele `POST` *parameters* bekend. Op dezelfde plaatst filtert de datacollectie ook de `GET` parameters uit de URL.
- De eventuele *gebruikersinteractie* is eveneens gekend na stap 3, door het opvragen van de `getHitTestResult` functie van het huidige WebView element. Deze functie kan echter enkel een klik op `<a>` objecten opvangen en geen indiening van formulieren. Er is geen manier gevonden om deze foutloos op te vangen waardoor de gebruikersdetectie maar gedeeltelijk werkt.
- De aanwezigheid en inhoud van *cookies* is eveneens afhankelijk van het type verzoek. Bij gewone verzoeken vraagt het `FrameLoader` object de cookies op



Figuur 5.3: Een vereenvoudigde voorstelling van het laden van een webpagina door de Android Browser.

na stap 6a. In het geval van een doorverwijzing vraagt het `RequestHandle` object deze op. Beide objecten maken gebruik van de `CookieManager` die voor de duidelijkheid te bewaren niet werd opgenomen in figuur 5.3.

- Het `LoadListener` object vangt het gebruik van de *HTTP authenticatie* methode op. De Android Browser verzendt de HTTP authenticatie niet automatisch met het eerste verzoek naar de server maar wacht op een `401` of `407` respons van de server. Bij ontvangst van dergelijk respons haalt dit object de informatie uit de database of vraagt ze de gegevens aan de gebruiker via de gebruikersinterface. Vervolgens voegt ze de HTTP authenticatie toe aan een nieuw verzoek naar de server. Deze manier van omgaan met HTTP authenticatie informatie verschilt van Mozilla. Mozilla stuurt namelijk sowieso authenticatie informatie mee, indien die beschikbaar is in de cache van de browser. Een bezoek van de testcases op `http://distrinet.cs.kuleuven.be/software/CsFire/csfiretests.php` met de Android Browser illustreert dit verschil. Ondanks ingelogd te zijn, stuurt de Android Browser de HTTP authenticatie informatie niet mee, omdat de specifieke testpagina's er niet expliciet naar vragen.

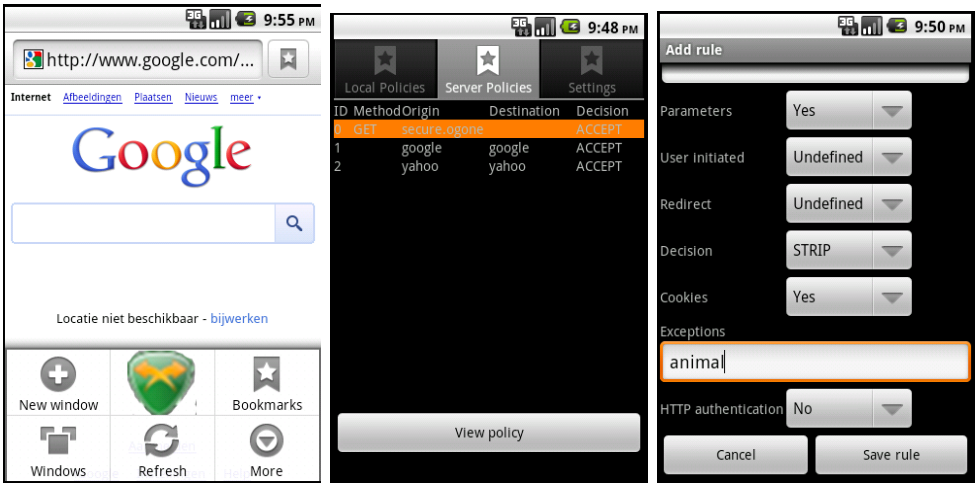
Beslissingsvorming

In CsDroid kan het vormen van de beslissing op twee verschillende plaatsen gebeuren (`RequestHandle` of `BrowserFrame` object). Beide objecten maken echter gebruik van hetzelfde `PolicyManager` object dat het beslissingsalgoritme bevat.

De `PolicyManager` controleert in dit geval ook weer op eventuele (lokale of server) uitzonderingen die zich in een SQLite database bevinden. De gebruiker kan de lokale uitzonderingen manueel invoeren zoals in de originele CsFire, en ook de server uitzonderingen kunnen van een vooraf bepaalde server gedownload worden. Figuur 5.4 toont enkele schermafbeeldingen van deze mogelijkheden. De klasse `ClientPolicy` implementeert tot slot het standaard beleid zoals voorgesteld in hoofdstuk 4.

Beslissingsoplegging

Indien de `PolicyManager` de `ACCEPT` beslissing neemt, onderneemt CsDroid geen verdere actie. In het geval van een `BLOCK` beslissing breekt CsDroid het verzoek af na stap 3, of in het geval van een doorverwijzing na stap 10a, waarna het systeem een foutmelding genereert. Bij de `STRIP` beslissing verwijdert CsDroid de cookies selectief via het `CookieManager` object. De verwijdering van de HTTP authenticatie hoofding gebeurt in het `LoadListener` object. CsDroid vraagt in dit geval een herinvoering van de authenticatie informatie door de gebruiker via het `CallBackProxy` object. De implementatie van de `ASK` beslissing bleek soms tot onverklaarbare crashes te leiden. Vandaar dat CsDroid deze beslissing, dewelke een minder belangrijk aspect vormt van het beslissingsalgoritme, niet ondersteunt.



Figuur 5.4: Enkele schermafbeeldingen van CsDroid.

5.2.2 Ondersteuning van CORS

Problematisch CORS gedrag

De Android documentatie biedt geen informatie over eventuele CORS ondersteuning, maar bepaalde sites suggereren wel dat de Android Browser de CORS specificatie volledig zou ondersteunen[8]. Na een grondige test van de browser kwamen echter volgende problemen aan het licht:

- Na enkele pogingen blijkt dat de browser maar éénmaal een CORS verzoek kan uitvoeren. Eens het eerste verzoek verzonden, kan noch datzelfde XHR object, noch een ander XHR object van op éénderwelke website, een CORS verzoek verzenden. Vanaf het tweede verzoek toont de browser op systematische wijze een nietszeggende foutmelding. Na lang zoeken en met behulp van enkele internet blogs, kon het probleem herleid worden tot de *Cache Manager* van de Android Browser [9]. Een gedeeltelijke oplossing om dit probleem te omzeilen, bevindt zich aan de server-zijde. Door aan elk antwoord de `Cache-Control: no-cache` hoofding mee te geven, slaat de browser het antwoord niet meer op en zijn achtereenvolgende CORS verzoeken wel mogelijk.

- Een tweede probleem betreft het gebruik van de `withCredentials` optie. Terwijl een CORS verzoek volgens de specificatie enkel sessie informatie mag toevoegen wanneer de waarde van de `withCredentials` gelijk is aan `true`, past de Android Browser dit niet toe. De browser voegt de sessie informatie daarentegen toe bij elk verzoek naar de server. Dit is zowel het geval bij CORS verzoeken zonder de correcte `withCredentials` waarde als bij voorafgaande

[8] http://caniuse.com/cors

[9] Volgende pagina biedt meer informatie over cache probleem van de Android Browser: http://mobilizejs.com/2011/03/20/android-webkit-xhr-status-code-0-and-expires-headers/

verzoeken. Een implementatie van het beslissingsalgoritme, zoals hoofdstuk 4 beschreef, zou dit probleem deels oplossen door deze informatie te verwijderen waar nodig.

- Een derde en laatste probleem betreft de CORS doorverwijzingen. Uit tests is gebleken dat het CORS mechanisme faalt vanaf de eerste CORS doorverwijzing, waardoor CsDroid geen rekening hoeft te houden met dit type verzoeken.

CORS database

CsDroid implementeert de CORS database, net zoals de lokale en server uitzonderingen, via de SQLite database. Ze bevat de velden zoals voorgesteld in tabel 4.1.

CORS verzoeken onderscheppen

De onderschepping van CORS verzoeken gebeurt via de controle van de *Origin header*. Er bleek geen éénvoudige manier te zijn om het XHR object zelf te onderscheppen. Momenteel is een controle van de *Origin header* echter voldoende omdat XHR objecten binnen de browser de enige objecten zijn die deze hoofding verzenden. Op figuur 5.3 bevindt deze controle zich na stap 3 in het `BrowserFrame` object.

Opvangen van antwoorden op voorafgaande verzoeken

Het opvangen van antwoorden op voorafgaande verzoeken gebeurt in het `LoadListener` object. Deze vangt de inkomende antwoorden op van de server en controleert op het gebruik van de `OPTIONS` methode. Indien het gaat om een respons met CORS informatie, slaat CsDroid de informatie op in de CORS database.

Uitvoeren van een voorafgaand verzoek

De Android Browser verzendt automatisch een voorafgaand verzoek indien het om een geavanceerd verzoek gaat. CsDroid dwingt echter ook voor eenvoudige verzoeken een voorafgaand verzoek af. De browser verzendt deze in het `BrowserFrame` object na stap 3 en slaat de verkregen informatie op in de CORS database, indien het om geldige CORS informatie gaat.

5.3 Conclusie

Dit hoofdstuk zette de aanpak van hoofdstuk 4 om in een implementatie voor twee mobiele browsers.

De ontwikkeling van CsFennec verliep redelijk vlot omdat de implementatie veel code van CsFire kon hergebruiken. Enkel de gebrekkige documentatie van Mozilla Fennec, de foutieve implementatie van de onderschepping van een klik door RequestPolicy en het onderscheppen van CORS verzoeken brachten de implementatie nu en dan in moeilijkheden. Niets was echter onoverkomelijk waardoor CsFennec alle

elementen van de voorgestelde aanpak bevat. Omdat een omzetting naar Mozilla Firefox zeer eenvoudig is, voorziet deze masterproef ook een aangepaste versie van CsFire.

Voor CsDroid zag het er lange tijd naar uit dat een CORS ondersteuning niet zou lukken, omwille van de gebrekkige ondersteuning door de Android Browser. Mits de inachtneming van volgende beperkingen, is de implementatie alsnog geslaagd. Vooreerst kan CsDroid enkel omgaan met een server die expliciet aangeeft dat de browser niets in de cache mag opslaan. Ten tweede is de `ASK` beslissing niet ondersteund in de uitzonderingen, al is dit niet zo heel belangrijk omdat de systeemvereisten reeds wezen op het belang van een autonoom beslissingsalgoritme. Ten derde verzendt CsDroid de HTTP authenticatie informatie enkel indien de website hier expliciet naar vraagt, ook al is deze beschikbaar in de cache van de browser. Ten vierde zal CsDroid sessie informatie gebruiken in een CORS verzoek wanneer de server dit toelaat maar de verzendende pagina dit niet aangaf via de `withCredentials` optie. Tot slot onderschept CsDroid enkel een klik op een link en dus geen klik op de invoerknop van een formulier. Voor het overige is de werking van CsDroid zoals de voorgestelde aanpak in het vorig hoofdstuk.

Volgend hoofdstuk bespreekt een verdere evaluatie van de implementaties en gaat dieper in op enkele bestaande websites die CORS ondersteuning aanbieden.

Hoofdstuk 6

Evaluatie

Dit hoofdstuk evalueert de in hoofdstuk 5 ontwikkelde extensies op 3 manieren. Ten eerste evalueert sectie 6.1 de werking van de CSRF basisbescherming. Verder gaat sectie 6.2 na of de extensies correct omgaan met een zelfgebouwd *Cross-Origin Resource Sharing* testplatform, dat voldoet aan de W3C specificaties. Tot slot onderzoekt sectie 6.3 in hoeverre de extensies kunnen omgaan met bestaande CORS ondersteunende websites.

6.1 Evaluatie van de CSRF bescherming

6.1.1 CsFennec

Het werk van De Ryck et. al. bevat reeds een grondige evaluatie van CsFire [48]. Dit zorgt ervoor dat ook een groot deel van de code van CsFennec reeds geëvalueerd is. Om na te gaan of CsFennec nog steeds dezelfde CSRF bescherming biedt, maakt deze evaluatie gebruik van een testkit die zich op de officiële website van CsFire bevindt[1]. Na controle blijkt alle functionaliteit nog te werken. De CsFire testkit omvat echter geen testen voor de gebruikersinteractie met formulieren en doorverwijzingen. Daarom vervolledigt een bijkomend testplatform de evaluatie door ook deze aspecten te behandelen. Ook deze functionaliteit blijkt na controle volledig te werken.

6.1.2 CsDroid

De implementatie van CsDroid vereist een uitgebreidere controle, aangezien deze aangepaste browser volledig zelf ontwikkeld is. De evaluatie voert testen uit met dezelfde testplatformen als CsFennec. Daarboven voert het nog eens extra testen uit met betrekking tot lokale en server uitzonderingen. Op deze wijze kwamen enkele foutjes aan het licht met betrekking tot het parsen van reguliere expressies en het toepassen van cookie uitzonderingen. Het opvangen van gebruikersinteractie loopt zoals verwacht niet helemaal perfect, de browser onderschept namelijk geen klik

[1] http://distrinet.cs.kuleuven.be/software/CsFire/csfiretests.php

op de invoerknop van een formulier. Zoals sectie 5.2 reeds aangaf, behandelt de Android Browser de HTTP authenticatie op een andere manier dan Mozilla. Om dit te kunnen testen, werd het eigen gebouwd testplatform uitgebreid met testen die de correcte verwijdering van HTTP authenticatie informatie nagaan. Nu en dan gebeurde het dat CsDroid tijdens een test schijnbaar willekeurig crashte. Uiteindelijk bleek het opvangen van een klik de reden hiervoor te zijn. Er werd voor gekozen deze functionaliteit te behouden omwille van de zeldzaamheid van deze crash, wat een verder onderzoek naar deze bug natuurlijk niet uitsluit.

6.1.3 Gebrek aan een evaluatie onder gebruikers

CsFennec en CsDroid hebben geen gebruikerstesten ondergaan. De eerste omwille van de beperkte ondersteuning door de huidige smartphone generatie, de tweede omwille van de moeilijke verspreiding van een aangepaste browser onder gebruikers. In het kader van deze evaluatie was dit niet haalbaar omwille van het beperkte tijdsbestek.

6.2 Evaluatie van de CORS ondersteuning

Deze sectie heeft als doel na te gaan in hoeverre CsDroid en CsFennec de werking van CORS toelaten. Ze vergelijkt de werking met de bestaande extensies die sectie 2.3.2 behandelde. Om de browser extensies grondig te kunnen testen is een testplatform ontwikkeld met specifieke CORS ondersteuning. Dit testplatform bevat vier CORS verzoeken waarvan twee eenvoudige verzoeken (EV) en twee geavanceerde verzoeken (AV). Voor elk type verzoek is er één met de `withCredentials` optie aan en één met deze optie uit. Tabel 6.1 geeft de resultaten weer van de testen. Ze bevat een evaluatie van drie desktop extensies en twee mobiele extensies, voor respectievelijk Mozilla Firefox en Mozilla Fennec. De overige testen gebeurden op basis van de beschrijving van de werking die beschreven stond in de bijhorende paper, omdat de extensies zelf niet beschikbaar waren.

6.2.1 Testen met bestaande extensies

NoScript 2.1.0.3

NoScript bestaat zowel voor Mozilla Firefox als Mozilla Fennec. Beiden hebben gemeenschappelijk dat ze standaard geen rekening houden met CORS verzoeken en zelfs alle JavaScript code blokkeren. Hierdoor houdt deze extensie elk CORS verzoek volledig tegen. Er zijn echter enkele belangrijke verschillen tussen beide versies.

De desktop versie bevat heel wat mogelijkheden tot verfijndere bescherming. Zo is er onder andere een component met de naam *Application Boundary Enforcer* die, zoals sectie 2.3.2 reeds aanhaalde, mogelijkheden tot CSRF bescherming aanbiedt. Op deze manier kan de gebruiker NoScript configureren zodat deze inter-domein verzoeken voor bepaalde sites toelaat of er enkel de authenticatie informatie van verwijdert. Een CORS detectie mechanisme ontbreekt echter waardoor NoScript hier geen

	EV	EV (credentials)	AV	AV (credentials)
NoScript	-	-	-	-
NoScript (ABE)	V	-	V	-
RequestPolicy	-	-	-	-
CsFire	V	-	V	-
AntiCSRF	V	/	V	/
CSRF Protector	-	-	-	-
CsFennec	V	V	V	V
CsDroid	V	V	V	V

TABEL 6.1: Evaluatie van de verschillende browser extensies aan de hand van een CORS testplatform (*V* wijst op volledige ondersteuning, / op gedeeltelijke ondersteuning en − op geen ondersteuning).

uitzondering voor kan maken. Omdat CORS verzoeken zonder de `withCredentials` optie geen gebruik maken van sessie informatie, is dit type verzoeken wel toegelaten indien de gebruiker onderstaande regels in de ABE invoert.

```
Site *
Anonymize GET POST
```

De mobiele versie van NoScript is heel wat beperkter dan de desktop versie. De Application Boundary Enforcer is dan ook niet beschikbaar op dit platform. De gebruiker kan enkel kiezen om CORS verzoeken volledig te blokkeren of volledig toe te laten. Zoals de bespreking van NoScript in sectie 2.3.2 reeds aangaf, bevat de mobiele versie zelfs geen enkele CSRF bescherming meer.

RequestPolicy 0.5.21

Zowel de desktop versie als de mobiele variant van RequestPolicy blokkeren alle inter-domein verzoeken, waardoor CORS ondersteuning is uitgesloten. Dit geldt voor elk type verzoek dat tijdens het testen aan bod kwam. De extensie laat enkel verzoeken toe na een toevoeging aan een lijst van uitzonderingen. De gebruiker dient dit echter telkens zelf manueel toe te laten of in te voeren, wat de gebruikerservaring sterk beïnvloedt. Bovendien verhuist de eindverantwoordelijkheid naar de gebruiker.

CsFire 0.9.2

CsFire stript alle verzoeken, behalve degene die de `GET` methode gebruiken zonder parameters en die veroorzaakt zijn door een klik op een link of een invoerknop van een formulier. Een CORS verzoek vertrekt steeds vanuit JavaScript code waardoor het standaard beleid deze verzoeken altijd ontdoet van sessie informatie. Dit zorgt ervoor dat enkel CORS verzoeken zonder de `withCredentials` optie aan, kunnen werken. Een optie om uitzonderingen te maken voor het CORS mechanisme is

niet beschikbaar. Net zoals bij RequestPolicy, kan de gebruiker wel uitzonderingen toevoegen zodat bepaalde CORS verzoeken alsnog werken.

BEAP (AntiCSRF)

AntiCSRF, een extensie die gebruik maakt van de *Browser-Enforced Authenticity Protection* methode, is niet teruggevonden op het internet waardoor de evaluatie deze niet in realiteit kon testen. De paper die de extensie introduceert, geeft geen expliciete CORS ondersteuning aan, al kan een gedeeltelijke ondersteuning wel afgeleid worden uit de beschreven aanpak [34]. Eenvoudige en geavanceerde verzoeken werken beide indien geen gebruik gemaakt wordt van de `withCredentials` optie. Dit omdat AntiCSRF, net zoals CsFire, geen verzoeken blokkeert maar enkel ontdoet van sessie informatie. Daarnaast laat deze extensie ook de CORS verzoeken met de `withCredentials` optie toe indien de website enkel cookies gebruikt. AntiCSRF stript namelijk geen cookies bij gebruik van het `HTTP` protocol. Desalniettemin vergroot op deze manier het gevaar voor CSRF aanvallen aangezien uit het CsFire onderzoek gebleken is dat websites vooral cookies gebruiken als authenticatie informatie. Net zoals alle vorige extensies kan AntiCSRF echter geen expliciete uitzonderingen maken voor CORS verzoeken, waardoor het dit systeem niet correct ondersteunt.

CSRF Protector

Net zoals BEAP is ook CSRF Protector niet meer beschikbaar voor installatie. Het beschreven beleid beschrijft echter geen specifieke CORS ondersteuning en blokkeert inter-domein verzoeken waardoor elk van de vier testen zou falen.

6.2.2 Testen met eigen extensies

CsFennec

Aangezien het CORS testplatform op een correcte manier omgaat met voorafgaande verzoeken, verbreekt CsFennec de CORS werking in geen van de vier testen. Een correcte behandeling van voorafgaande verzoeken is nodig opdat ook eenvoudige CORS verzoeken zouden werken. De ontwikkelde aanpak verplicht namelijk ook expliciete toelating voor eenvoudige verzoeken, in tegenstelling tot de W3C specificatie van CORS. Op deze manier ontstaat een systeem waarbij servers op een dynamische manier uit het standaard beleid van CsFennec kunnen stappen. Omwille van het belang van een correcte behandeling van voorafgaande CORS verzoeken, gaat volgende sectie onder andere na in hoeverre bestaande websites voorafgaande CORS verzoeken correct behandelen.

CsDroid

Ook CsDroid ondersteunt de CORS werking, al moet men wel rekening houden met de bestaande problemen van de Android Browser. Zoals sectie 5.3 aangaf, moet een CORS server in de hoofding van zijn antwoord meegeven dat het antwoord niet

gecached mag worden. Ten tweede zal CsDroid altijd sessie informatie verzenden wanneer het voorafgaand verzoek dit toelaat. Stel dat een website de `withCredentials` functie niet aanzet, dan nog zal de Android Browser sessie informatie trachten te verzenden. CsDroid blokkeert deze sessie informatie enkel wanneer het voorafgaand verzoek sessie informatie verbiedt, waarmee de werking dus gedeeltelijk afwijkt van CsFennec. Deze masterproef gaat er echter vanuit dat dit niet veel veiligheidsproblemen creëert omdat de webpagina er evengoed voor kon kiezen om wel sessie informatie mee te sturen. Het antwoord van de server primeert dus in alle gevallen.

6.3 Evaluatie van CORS ondersteunende websites

Dit laatste deel van de evaluatie test de extensies aan de hand van bestaande websites die gebruik maken van het CORS mechanisme. Een zoektocht naar CORS ondersteunende mechanismen resulteerde in de conclusie dat erg weinig websites dit systeem ondersteunen. Dit is echter normaal omwille van de prille leeftijd van CORS. Uiteindelijk onderzoekt deze evaluatie 30 websites die een zekere vorm van CORS aanbieden. De CORS ondersteunende websites werden ontdekt via de website `http://enabled-cors.org`, een crawl van de 10000 meest bezochte websites[2] en een crawl van de 1000 populairste mashup API's[3] .

Tabellen A.1 en A.2, die zich in bijlage A bevinden, bieden een overzicht van de evaluatie van de 30 onderzochte websites. Volgende opsomming vat de belangrijkste conclusies samen:

- **Veelvuldig gebruik van `Access-Control-Allow-Origin:*`**

  De meeste websites maken gebruik van CORS om universele toegang tot data mogelijk te maken (27 van de 30 websites). Dit komt hoofdzakelijk omdat het grootste deel van de geteste websites enkel als doel heeft informatieverspreiding mogelijk te maken (bv. woordenboeken). Het gebruik van cookies of de HTTP authenticatie methode is hier sowieso uitgesloten. Sectie 3.2.1 wees er namelijk op dat sessie informatie niet toegestaan is in combinatie met een ACAO waarde gelijk aan `"*"`.

- **Goede ondersteuning van voorafgaande verzoeken**

  Een correcte ondersteuning van voorafgaande verzoeken is van groot belang voor CsFennec en CsDroid. Stel dat een website eenvoudige CORS verzoeken met sessie informatie toelaat maar geen voorafgaande verzoeken ondersteunt. Dan is de extensie niet op de hoogte van de toelating en verwijdert het de eventueel aanwezige sessie informatie. De evaluatie gaat daarom na in hoeverre de onderzochte websites voorafgaande CORS verzoeken ondersteunen. Uiteindelijk blijken 29 van de 30 websites een correct antwoord te geven op een voorafgaand verzoek. In de meeste gevallen kan de extensie de CORS ondersteuning van

[2] De lijst van meest bezochte websites werd verkregen van `http://www.alexa.com`.
[3] De lijst van de meest gebruike API's werd verkregen van `http://www.programmableweb.com`

een website dus te weten komen via een dergelijk verzoek om er later rekening mee te houden in het beslissingsalgoritme.

- **Beperkt gebruik van geavanceerde CORS verzoeken en sessie informatie**

  Ondanks dat de meeste websites voorafgaande verzoeken correct beantwoorden, is de informatie die hierin vervat meestal beperkt tot de `ACAO` hoofding. Er zijn maar twee websites teruggevonden die gebruik maken van de gevorderde CORS mogelijkheden (toelaten sessie informatie en geavanceerde verzoeken). De evaluatie is hier dus in zekere zin beperkt tot eenvoudige verzoeken zonder gebruik van sessie informatie. Een herevaluatie zal later moeten plaatsvinden wanneer meerdere websites gebruik maken van de volledige waaier van CORS mogelijkheden.

- **Gebrekkige ondersteuning van de `Access-Control-Allow-Methods` hoofding**

  De `Access-Control-Allow-Methods` hoofding geeft normaal gezien aan welke methodes een CORS verzoek mag gebruiken. De meeste voorafgaande verzoeken maken daarentegen wel gebruik van de `Allow` hoofding. Deze hoofding bevat eveneens een opsomming van de toegelaten methodes en kan daarom eventueel als vervanger voor de `ACAM` hoofding fungeren. Om tegemoet te komen aan deze eigenschap, werd een aanpassing doorgevoerd aan de extensies door de `Allow` informatie op te slaan bij afwezigheid van de `ACAM` hoofding.

6.4 Conclusie

Dit hoofdstuk beschreef de uitvoerige evaluatie van de belangrijkste aspecten van de ontwikkelde extensies. Naast de succesvolle evaluatie van de basisbescherming en de ondersteuning van een zelfgebouwd CORS platform, ging dit hoofdstuk na in hoeverre de CORS ondersteunende websites de W3C specificatie volgen. De extensies kunnen de CORS ondersteuning bijna altijd correct afleiden uit het voorafgaand verzoek. Dit omdat CORS servers ook voorafgaande verzoeken ondersteunen, ondanks dat ze enkel eenvoudige verzoeken toestaan. De waarneming dat de `Access-Control-Allow-Methods` hoofding vaak niet voorkwam, maar wel de `Allow` hoofding, leidde tot een aanpassing van de extensies. De datacollectie houdt nu namelijk ook rekening met de laatstgenoemde hoofding voor het toelaten van methodes.

Hoofdstuk 7

Besluit

7.1 Een overzicht van het geleverde werk

Deze masterproef voerde onderzoek naar het beveiligingsprobleem *Cross-Site Request Forgery* op mobiele apparaten. De doelstellingen die de inleiding vooropstelde, waren enerzijds een grondig onderzoek naar de veiligheid van inter-domein communicatie en anderzijds de ontwikkeling van een CSRF beschermingsmaatregel voor mobiele browsers. Deze sectie zet de verschillende bijdragen nog eens op een rij per doelstelling.

7.1.1 Onderzoek naar de veiligheid van inter-domein communicatie

Een eerste bijdrage bestaat bestaat uit het onderzoek naar de huidige veiligheidsproblematiek van inter-domein communicatie.

Het begrip *Cross-Site Request Forgery* werd in deze masterproef uitgebreid behandeld waarbij gewezen werd op de risico's en de grote impact die een CSRF aanval met zich kan meedragen. Uiteindelijk bleek dat een bescherming aan de client-zijde noodzakelijk is omwille van de vele onbeschermde websites die aanwezig zijn op het internet. Uit onderzoek naar bestaande extensies aan de client-zijde kwam CsFire naar voren als maatregel die de beste balans biedt tussen doeltreffendheid en een ongeschonden gebruikerservaring. De schaarse extensies voor mobiele browsers konden geen gelijkaardige bescherming bieden, waardoor een implementatie van CsFire voor deze mobiele platformen zich opdrong.

Vervolgens werden enkele recent ontwikkelde browser mechanismen onderzocht, die veilige inter-domein communicatie trachten mogelijk te maken. Browsers blijken momenteel de beste ondersteuning te bieden voor CORS, wat staat voor *Cross-Origin-Resource Sharing*. Na een grondig onderzoek bleek CORS echter te kampen met enkele CSRF problemen. Eenvoudige CORS verzoeken zijn namelijk niet beschermd tegen CSRF aanvallen en indien een ontwikkelaar zich niet aan het DBAD design houdt, blijft het risico op CSRF-achtige aanvallen bestaan bij elk type CORS verzoek. Een ander inter-domein mechanisme, het *Uniform Messaging Policy*, pakt deze twee problemen aan door geen gebruik meer te maken van de bestaande sessie mechanismen

(cookies en HTTP authenticatie). Momenteel ondersteunt echter geen enkele browser dit mechanisme.

7.1.2 Ontwikkeling van een CSRF bescherming voor mobiele browsers

Een tweede bijdrage betreft de ontwikkeling van een CSRF beveiligingsmaatregel voor browsers op mobiele platformen. Dit resulteerde enerzijds in CsFennec, de eerste extensie voor Mozilla Fennec die een CSRF beveiliging aanbiedt en rekening houdt met een zo klein mogelijke impact op web 2.0 toepassingen. Anderzijds levert deze bijdrage ook CsDroid, een aangepaste versie van de Android Browser en daarmee de eerste Android Browser die een CSRF bescherming bevat.

De derde bijdrage bestaat uit een combinatie van de eerste twee bijdragen. De resultaten die uit de eerste bijdrage volgden, leidden tot de conclusie dat momenteel geen enkele CSRF bescherming rekening houdt met de mogelijkheden van *Cross-Origin Resource Sharing*. De bestaande extensies blokkeren de werking ervan gedeeltelijk of zelfs helemaal. Vandaar dat een aanpak ontwikkeld werd met aanpassingen voor CsFennec en CsDroid die de werking van CORS volledig toelaten. Bovendien gaat de aanpak zelfs een stap verder door het originele CORS mechanisme aan te passen om op die manier een betere CSRF bescherming te kunnen bieden. Deze aanpassing houdt in dat de browser ook voor eenvoudige verzoeken een voorafgaand CORS verzoek verzendt. Indien blijkt dat de server het gebruik van sessie informatie niet toelaat, verwijdert de browser deze informatie uit het eenvoudig CORS verzoek. CsFennec en CsDroid zijn daarmee de eerste CSRF extensies die rekening houden met het bestaan van het CORS mechanisme en er handig gebruik van maken om websites toe te laten uit het standaard beslissingsalgoritme te stappen.

7.2 Voor- en nadelen van de voorgestelde oplossing

Deze sectie zet de voor- en nadelen van de ontwikkelde beveiligingsmaatregelen nog eens op rij.

7.2.1 Voordelen

- CsFennec en CsDroid bouwen voort op de resultaten van het onderzoek rond CsFire. Deze resultaten hebben tot een beslissingsalgoritme geleid dat een goede balans verzorgt tussen een doeltreffende CSRF bescherming en een ongeschonden gebruikerservaring. Door deze werking over te nemen, bezitten de mobiele extensies over dezelfde positieve eigenschappen.
- CsFennec en CsDroid zijn de eerste CSRF maatregelen die rekening houden met de CORS werking en dit mechanisme niet geheel of gedeeltelijk blokkeren.
- Browsers bieden websites via CORS de mogelijkheid om uit het *Same Origin Policy* te stappen. CsFennec en CsDroid maken hiervan handig gebruik om ook uit de standaardwerking van de CSRF bescherming te stappen. Op deze

manier realiseren de extensies een beslissingsalgoritme dat gebruik maakt van uitzonderingen die websites op een dynamische manier kunnen toevoegen.

- De extensie is ontwikkeld voor Mozilla Firefox, Mozilla Fennec en de Android Browser waardoor ze één desktopplatform en twee mobiele platformen ondersteunt. Het feit dat nu ook mobiele gebruikers van de bescherming kunnen genieten, is een grote stap voorwaarts omwille van het stijgende aantal van dit type gebruikers.

7.2.2 Nadelen

- De voorgestelde oplossing verplicht web ontwikkelaars nog steeds om het *Don't Be A Deputy* (DBAD) ontwerp toe te passen. Indien dit niet gebeurt, lopen web applicaties nog steeds het gevaar slachtoffer te worden van CSRF-achtige aanvallen.

- Doorverwijzingen vormen een probleem voor CsFire omdat deze geen rekening hield met de aanwezigheid van eventuele *Open Redirects*, of open doorverwijzingen. CsFennec en CsDroid kampen met dezelfde problemen omdat ze deze werking rechtstreeks overnemen. Een eventuele oplossing bestaat uit het toevoegen van alle betrokken domeinen in de *Referrer Header* of *Origin Header*, maar de huidige browsers ondersteunen deze techniek nog niet.

- De evaluatie van de CORS ondersteunende websites leidde tot de conclusie dat het CORS mechanisme nog geen veelvuldig gebruik kent. Momenteel kunnen dus niet veel websites profiteren van deze functionaliteit.

- Ondanks de poging een minimale impact op de gebruikerservaring te veroorzaken, verliezen sommige websites alsnog een deel van hun functionaliteit. Wanneer een website bijvoorbeeld gebruik maakt van de Facebook API om na te gaan welke Facebook vrienden van de gebruiker fan zijn van de website, zal de extensie de cookies verwijderen in het inter-domein verzoek. Hierdoor weet Facebook niet om welke gebruiker het gaat en kan deze het verzoek niet beantwoorden.

- Omdat de Android Browser geen platform aanbiedt om extensies te ontwikkelen, moesten aanpassingen in de broncode van de browser gebeuren. Deze aanpassingen dienden zelfs in de Webkit bibliotheek te gebeuren die ingebouwd zit in het Android platform. CsDroid is bijgevolg niet makkelijk te verspreiden onder gebruikers.

7.3 Toekomstige aanpassingen en uitbreidingen

Deze laatste sectie somt enkele slotbedenkingen op die volgden uit het inzicht dat tijdens het verloop van deze masterproef verkregen werd.

- Een eerste punt betreft de behandeling van doorverwijzingen. Indien browsers de meeste recente specificatie van de *Origin Header* toepassen, zal een nieuwe versie van de extensie hiermee rekening moeten houden.
- Momenteel maken CsFennec en CsDroid gebruik van CORS informatie die zich op de server bevindt. Een aangepaste extensie zou, gelijkaardig aan CSRF Protector, ook het `crossdomain.xml` bestand van Adobe, of het `clientaccess-policy.xml` bestand van Silverlight kunnen opvragen. Op deze manier is nog meer informatie bekend over het toegelaten inter-domein verkeer.
- Een aangepaste extensie zou meer voordeel kunnen halen uit de beschikbare browsercontext. Op een gelijkaardige manier als het onderzoek van Shahriar en Zulkernine zou een extensie zich bijvoorbeeld kunnen baseren op de open tabbladen in de browser.
- Een toekomstige extensie moet sowieso de nieuwe stand van zaken betreffende de CORS ondersteuning van websites in rekening brengen. Dit kan gepaard gaan met een uitgebreide herevaluatie van de hier ontwikkelde extensies door de werking te controleren van de nieuwe CORS ondersteunende websites.

Tot slot volgt nog een laatste bemerking met betrekking tot veilig inter-domein verkeer. De redenen die aan de basis liggen van *Cross-Site Request Forgery*, zijn ontstaan door het ondoordacht ontwerp van bepaalde browsersystemen: het Same Origin Policy en het inter-domein gebruik van cookies of de HTTP authenticatie methode. Omdat miljoenen websites zich hierop gebaseerd hebben, is het simpelweg aanpassen van deze systemen niet haalbaar in een korte tijdspanne. Websites moeten er echter naar streven veiligere authenticatie mechanismen te gebruiken. Een belangrijk mechanisme dat websites het gebruik van een dergelijk systeem oplegt, is het *Uniform Messaging Policy*, een inter-domein communicatie mechanisme dat hoofdstuk 3 kort behandelde. Tot de dag dat websites zich hieraan hebben aangepast, zullen extensies zoals CsFennec en CsDroid onvermijdbaar zijn, om de gebruiker te beschermen tegen ongeoorloofd inter-domein verkeer.

Bijlage A

Een evaluatie van CORS ondersteunende websites

Tabellen A.1 en A.2 geven de resultaten weer van een evaluatie van 30 CORS ondersteunende websites. Ze gaat voor elke website volgende zaken na:

- *Ondersteuning voorafgaand verzoek:* Indien een cel de waarde `V` bevat, stuurt de server een antwoord op een voorafgaan verzoek en kan de CORS ondersteuning uit dit antwoord worden afgeleid. Zo niet bevat de cel de waarde `-`.
- *EV ondersteuning:* Deze kolom geeft aan of een website gebruik maakt van eenvoudige verzoeken. Indien dit het geval is, bevat een cel de waarde `V`, zo niet de waarde `-`.
- *AV ondersteuning:* Deze kolom geeft aan of een website gebruik maakt van geavanceerde verzoeken. Indien dit het geval is, bevat een cel de waarde `V`, zo niet de waarde `-`.
- *ACAC hoofding:* Bevat de waarde van het `Access-Control-Allow-Credentials` veld dat zich in het antwoord op het voorafgaand verzoek bevindt.
- *ACAO hoofding:* Bevat de waarde van het `Access-Control-Allow-Origin` veld dat zich in het antwoord op het voorafgaand verzoek bevindt. Dit is ofwel de waarde `"*"` ofwel de waarde `"origin"` die de origine van de verzender bevat (indien door de server toegelaten).
- *ACAM hoofding:* Geeft aan of het `Access-Control-Allow-Methods` veld een correcte waarde bevat.
- *ALLOW hoofding:* Geeft aan of het `Allow` veld een correcte waarde bevat.
- *ACMA hoofding:* Bevat de waarde van het `Access-Control-Max-Age` veld dat zich in het antwoord op het voorafgaand verzoek bevindt.

Website	Ondersteuning voorafgaand verzoek	EV ondersteuning	AV ondersteuning	ACAC hoofding
sourceforge.net	V	V	-	-
naver.jp	V	V	V	true
sarenza.com	V	V	-	-
adopteunmec.com	V	V	-	-
feedly.com	V	V	-	-
mademan.com	V	V	-	-
boostmobile.com	V	V	-	-
themefuse.com	V	V	-	-
needforspeed.com	V	V	-	-
centraldesktop.com	V	V	-	-
bethsoft.com	V	V	-	-
intermedia.net	V	V	-	-
buzzword.org.uk	V	V	-	-
xmlns.com	V	V	-	-
freeclass.eu	V	V	-	-
heppnetz.de	V	V	-	-
rdfs.org	V	V	-	-
ontologi.es	V	V	-	-
semantic.eurobau.com	V	V	-	-
bibliographica.org	V	V	-	-
ckan.net	V	V	-	-
data-gov.tw.rpi.edu	V	V	-	-
logd.tw.rpi.edu	V	V	-	-
geonames.org	-	V	-	-
productdb.org	V	V	-	-
rkbexplorer.com	V	V	-	-
semanticweb.org	V	V	-	-
sameas.org	V	V	-	-
prefix.cc	V	V	-	-
enabled-cors.org	V	V	-	-

Tabel A.1: Een uitgebreide evaluatie van 30 CORS ondersteunende websites.

Website	ACAO hoofding	ACAM hoofding	ALLOW hoofding	ACMA hoofding
sourceforge.net	*	-	-	-
naver.jp	origin	V	-	1728000
sarenza.com	*	-	V	-
adopteunmec.com	*	-	V	-
feedly.com	*	-	V	-
mademan.com	*	-	-	-
boostmobile.com	https:// apps.boost mobile.com	-	V	-
themefuse.com	*	-	-	-
needforspeed.com	*	-	-	-
centraldesktop.com	*	-	-	-
bethsoft.com	*	-	V	-
intermedia.net	*	-	V	-
buzzword.org.uk	*	-	V	-
xmlns.com	*	-	V	-
freeclass.eu	*	-	V	-
heppnetz.de	*	-	V	-
rdfs.org	*	-	V	-
ontologi.es	*	-	-	-
semantic.eurobau.com	*	-	V	-
bibliographica.org	*	-	-	-
ckan.net	*	-	-	-
data-gov.tw.rpi.edu	*	-	-	-
logd.tw.rpi.edu	*	-	V	-
geonames.org	-	-	V	-
productdb.org	*	-	-	-
rkbexplorer.com	*	-	-	-
semanticweb.org	*	-	-	-
sameas.org	*	-	-	-
prefix.cc	*	-	-	-
enabled-cors.org	*	-	-	-

Tabel A.2: Een uitgebreide evaluatie van 30 CORS ondersteunende websites (vervolg).

Bijlage B

Documentatie bij het gebruik van de extensies

B.1 CsFennec

De versie van Mozilla Fennec met de vooraf geïnstalleerde CsFennec extensie, kan teruggevonden worden op de live-dvd die zich in appendix E bevindt. Volgende subsecties lichten het gebruik van CsFennec verder toe.

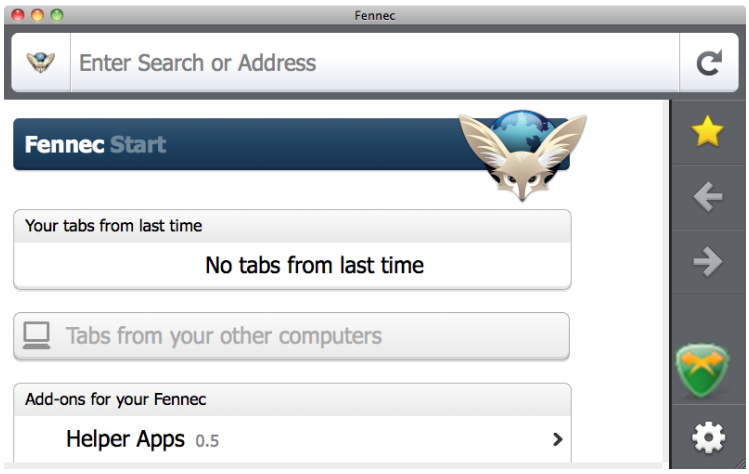
B.1.1 Opstarten

Bij het opstarten van de browser is CsFennec reeds ingeschakeld. Mozilla Fennec heeft zowel links als rechts extra knoppen voorzien, al zijn deze standaard niet zichtbaar. Om ze zichtbaar te maken dient men de rechter muisknop in te houden in het hoofdscherm en een beweging naar links of rechts te maken. Aan de rechterzijde kan men het CsFennec logo waarnemen dat een groene kleur heeft indien CsFennec aanstaat. Een klik op dit logo opent het menu met verschillende opties.

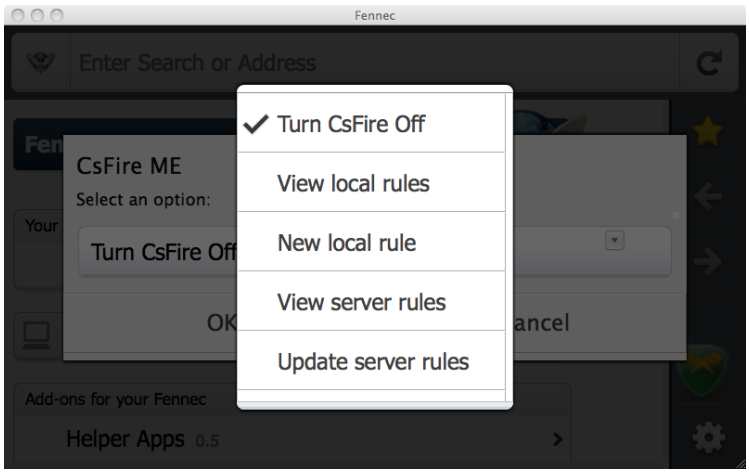
B.1.2 Gebruik van het menu

Het menu bestaat uit een *drop-down* lijst en bevat verschillende mogelijkheden. Een optie kan geladen worden door hem te selecteren en op OK te klikken.

- *Turn CsFire off/on:* Hiermee kan de extensie aan of uit gezet worden.
- *View local rules:* Deze optie laat toe de lokale uitzonderingen te bekijken.
- *New local rule:* Een nieuwe lokale uitzondering kan via deze optie toegevoegd worden.
- *View server rules:* Deze optie laat toe de server regels te bekijken.
- *Update server rules:* Via deze knop controleert de extensie of er nieuwe uitzonderingen beschikbaar zijn op de server.



Figuur B.1: Het CsFennec logo bevindt zich aan de rechterkant van het hoofdscherm in Mozilla Fennec.



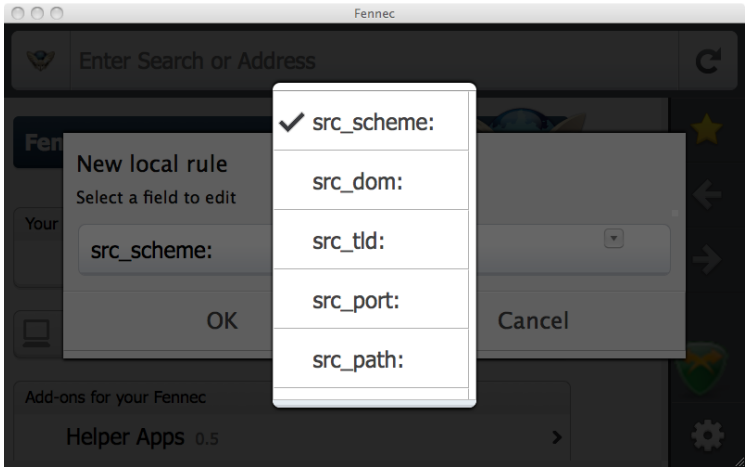
Figuur B.2: Het hoofdmenu van CsFennec.

- *Change server settings:* Hiermee kan de het adres van de server met uitzonderingen bekeken en aangepast worden.

- *View log:* Deze laatste optie toont een scherm met daarin een overzicht van de recente acties van de extensie.

B.1.3 Toevoegen van een lokale uitzondering

Het toevoegen van een lokale uitzondering vereist wat meer uitleg. Deze sectie legt uit welke opties beschikbaar zijn en hoe een uitzondering toegevoegd kan worden.



FIGUUR B.3: Het submenu van CsFennec dat toelaat een lokale uitzondering toe te voegen.

Eigenschappen origine

- *src_scheme*: Het gebruikte protocol. (bv. `http`, `https`...)
- *src_dom*: Het domein zonder de domein extensie. (bv. `google`, `www.google`...)
- *src_tld*: De extensie van de origine. (bv. `be`, `com`, `net`...)
- *src_port*: De poort die gebruikt wordt. (bv. `80`)
- *src_path*: Het pad van de webpagina die het verzoek verzendt. (bv. `/data/index.html`)

Eigenschappen bestemming

Dezelfde opsomming als bij de eigenschappen van de origine is hier van toepassing. Al hebben ze in dit geval betrekking op de bestemming van het verzoek. In plaats van het prefix `src_` bezitten ze in dit geval het prefix `dst_`.

Algemene eigenschappen van het verzoek

- *method*: De gebruikte methode van het verzoek. (bv. `GET`, `POST`...)
- *params*: Het al dan niet gebruik van parameters.
- *user*: Geeft aan of het verzoek geïnitieerd werd door de gebruiker. (bv. klik op link)
- *redirect*: Geeft aan of het verzoek het gevolg is van een doorverwijzing.

Eigenschappen beslissing

Deze eigenschappen hebben betrekking op de beslissing die uitgevoerd moet worden indien voldaan werd aan de voorwaarden die in de voorgaande eigenschappen beschreven werden:

- *decision*: De algemene beslissing van de regel. (`ACCEPT`, `STRIP`, `BLOCK` of `ASK`)
- *strip_cookies*: Geeft aan of ook cookies moeten gestript worden. (Enkel mogelijk indien de algemene beslissing `STRIP` is)
- *cookies*: Geeft aan welke uitzonderingen er zijn op de gestripte cookies. (Enkel mogelijk indien de algemene beslissing `STRIP` is en de optie *Strip Cookies* werd aangeduid)
- *strip_httpauth*: Geeft aan of de HTTP authenticatie hoofding moet gestript worden. (Enkel mogelijk indien de algemene beslissing `STRIP` is)
- *httpauth*: Deze optie heeft geen nut meer in de laatste versie van CsFennec en kan genegeerd worden.

Eens alle zaken correct zijn ingesteld, kan de gebruiker op `cancel` klikken. Vervolgens vraagt de extensie of de uitzondering al dan niet mag opgeslagen worden. Een uitgebreid voorbeeld dat de werking hiervan illustreert, kan teruggevonden worden op het testplatform dat standaard opgestart wordt met de meegeleverde Mozilla Fennec (zie live dvd).

B.2 CsDroid

CsDroid bevindt zich net zoals CsFennec op de live-dvd die zich in appendix E bevindt.

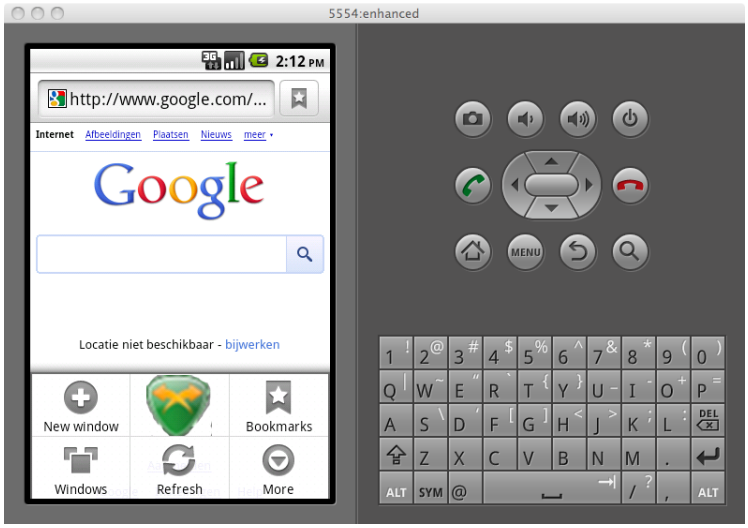
B.2.1 Opstarten

Een eerste stap bestaat uit het opstarten van de Android emulator. Dit kan door op het bureaublad van de live-dvd de snelkoppeling CsDroid aan te klikken. Vervolgens kan de browser opgestart worden via het wereldbol icoon. Bij het opstarten van de browser is de CSRF beveiliging reeds ingeschakeld. Door op de menuknop te klikken, verschijnt het browsermenu beneden in de applicatie. Hier bevindt zich het CsDroid logo dat een groene kleur heeft indien de beveiliging aanstaat. Een klik op dit logo opent het menu met de verschillende opties.

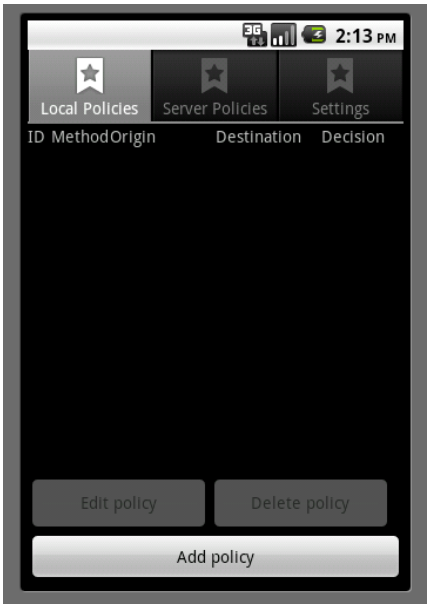
B.2.2 Gebruik van het menu

Het menu bestaat uit 3 tabbladen die elk een bepaalde groep van functies verzamelen.

- *Local Policies:* Via dit tabblad kunnen lokale uitzonderingen bekeken, aangepast en toegevoegd worden.



FIGUUR B.4: Het Android Browser opgestart via de Android Emulator.



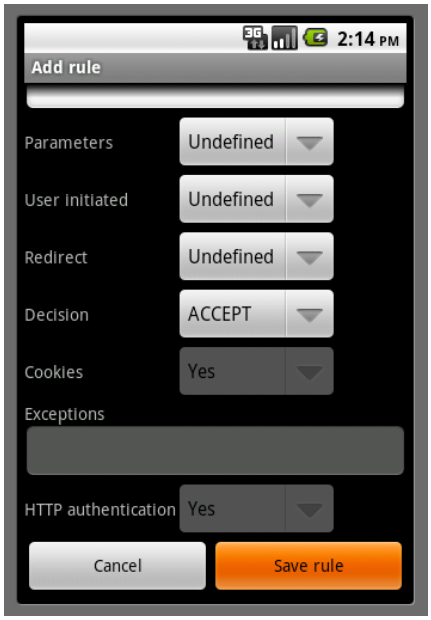
FIGUUR B.5: Het hoofdmenu van CsDroid dat uit 3 tabbladen bestaat.

- *Server Policies:* Dit tabblad toont de verschillende uitzonderingen die gedownload werden van de server.

- *Settings:* Dit tabblad laat toe de beveiliging uit te zetten en de server uitzonderingen te downloaden.

B.2.3 Toevoegen van een lokale uitzondering

Het toevoegen van een lokale uitzondering vereist wat meer uitleg. Deze sectie legt uit welke opties beschikbaar zijn en hoe een uitzondering toegevoegd kan worden.



Figuur B.6: Het submenu van CsDroid dat toelaat een lokale uitzondering toe te voegen.

Eigenschappen origine

- *src_scheme*: Het gebruikte protocol. (bv. `http`, `https`...)

- *src_dom*: Het domein zonder de domein extensie. (bv. `google`, `www.google`...)

- *src_tld*: De extensie van de origine. (bv. `be`, `com`, `net`...)

- *src_port*: De poort die gebruikt wordt. (bv. `80`)

- *src_path*: Het pad van de webpagina die het verzoek verzendt. (bv. `/data/index.html`)

Eigenschappen bestemming

Dezelfde opsomming als bij de eigenschappen van de origine is hier van toepassing. Al hebben ze in dit geval betrekking op de bestemming van het verzoek. In plaats van het prefix `src_` bezitten ze in dit geval het prefix `dst_`.

Algemene eigenschappen van het verzoek

- *Method*: De gebruikte methode van het verzoek. (bv. `GET`, `POST`...)
- *Parameters*: Het al dan niet gebruik van parameters.
- *User initiated*: Geeft aan of het verzoek geïnitieerd werd door de gebruiker. (bv. klik op link)
- *Redirect*: Geeft aan of het verzoek het gevolg is van een doorverwijzing.

Eigenschappen beslissing

Deze eigenschappen hebben betrekking op de beslissing die uitgevoerd moet worden indien voldaan werd aan de voorwaarden die in de voorgaande eigenschappen beschreven werden:

- *Decision*: De algemene beslissing van de regel. (`ACCEPT`, `STRIP`, `BLOCK` of `ASK`)
- *Cookies*: Geeft aan of ook cookies moeten gestript worden. (Enkel mogelijk indien de algemene beslissing `STRIP` is)
- *Exceptions*: Geeft aan welke uitzonderingen er zijn op de gestripte cookies. (Enkel mogelijk indien de algemene beslissing `STRIP` is en de optie *Cookies* werd aangeduid)
- *HTTP authentication*: Geeft aan of de HTTP authenticatie hoofding moet gestript worden. (Enkel mogelijk indien de algemene beslissing `STRIP` is)

Eens alle zaken correct zijn ingesteld, kan de gebruiker op `Save rule` klikken om de uitzondering op te slagen. Een uitgebreid voorbeeld dat de werking hiervan illustreert, kan teruggevonden worden op het testplatform dat standaard opgestart wordt met de meegeleverde CsDroid (zie live dvd).

Bijlage C

Populariserend artikel

Het Engelstalige populariserende artikel dat voor deze masterproef geschreven werd, bevindt zich in het bestand `bijlage_populariserend_artikel.pdf`.

Bijlage D

Wetenschappelijk artikel

Het Engelstalige wetenschappelijke artikel dat voor deze masterproef geschreven werd, bevindt zich in het bestand `bijlage_wetenschappelijk_artikel.pdf`.

Bijlage E

Live-DVD met CsDroid en CsFennec

De DVD die zich op deze pagina bevindt, is een live-dvd met een Ubuntu besturingssysteem waarop de ontwikkelde extensies zich bevinden. Op het bureaublad bevindt zich een `README` bestand dat de verdere instructies bevat.

Bijlage F

DVD met source code

De DVD die zich op deze pagina bevindt, bevat de source code van de ontwikkelde extensies. Daarnaast is er ook een `README` bestand dat de verdere instructies bevat.

Bibliografie

[1] Facebook touch. http://touch.facebook.com/.

[2] How to defeat digg.com. http://4diggers.blogspot.com/, Juni 2006.

[3] Csrf: Flash + 307 redirect = game over. http://lists.webappsec.org/pipermail/websecurity_lists.webappsec.org/2011-February/007533.html, February 2010.

[4] Defining safer json-p. http://www.json-p.org/, 2011.

[5] Introducing json. http://www.json.org/, 2011.

[6] Noscript. http://noscript.net/, 2011.

[7] Adobe. Allowing cross-domain data loading. http://livedocs.adobe.com/flash/9.0/main/wwhelp/wwhimpl/common/html/wwhelp.htm?context=LiveDocs_Parts&file=00001085.html.

[8] Adobe. Actionscript 3.0 reference for the adobe flash platform. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/URLRequestHeader.html, 2011.

[9] D. Akhawe, A. Barth, P. E. Lam, and J. Mitchell. Towards a formal foundation of web security. *Computer Security Foundations Symposium (CSF)*, Juli 2010.

[10] Apple. Webkit. http://www.webkit.org.

[11] M. Austin. Hacking facebook with html5. http://m-austin.com/blog/?p=19, July 2010.

[12] A. Bailey. Google mail exploit. http://www.cyber-knowledge.net/blog/2007/01/01/gmail-vulnerable-to-contact-list-hijacking, 2007.

[13] A. Barth. The web origin concept. http://tools.ietf.org/html/draft-abarth-origin-09, 2010.

[14] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. *CCS*, 2008.

[15] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2010Ð2015. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.pdf, February 2011.

[16] T. W. A. S. Consortium. Wasc threat classification: Cross-site request forgery. http://projects.webappsec.org/w/page/13246919/Cross-Site-Request-Forgery, 2009.

[17] H. Dwivedi, C. Clarck, and D. Thiel. *Mobile application security*. McGraw-Hill, 1st, edition, 2010.

[18] ebursztein. The absence of synchronous message api make impossible to pass options to scripts that are loaded before the page to block content. http://code.google.com/p/chromium/issues/detail?id=54257, September 2010.

[19] R. Fielding and T. B. Lee. Hypertext transfer protocol. http://tools.ietf.org/html/rfc2616, 1999.

[20] Finextra. M-banking set to grow in western europe. http://www.finextra.com/news/fullstory.aspx?newsitemid=18445, 2008.

[21] Google. Google web toolkit. http://code.google.com/webtoolkit/, 2011.

[22] J. Grossman. Csrf: The sleeping giant. 2006.

[23] J. Grossman. Csrf: The sleeping giant of website vulnerabilities. *RSA Conference*, 2008.

[24] Hackers.org. Xss (cross site scripting) cheat sheet. http://ha.ckers.org/xss.html.

[25] N. Hamiel and S. Moyer. Dynamic cross-site request forgery: A per-request approach to session riding. 2009.

[26] N. Hardy. The confused deputy. *ACM SIGOPS Operating Systems Review*, 22(4), October 1988.

[27] M. Heiderich. Csrfx. http://code.google.com/p/csrfx/.

[28] C. Jackson. Cors redirect behavior proposal. http://www.mail-archive.com/public-webapps@w3.org/msg05347.html, September 2009.

[29] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2006.

[30] J. Kemp. Cors, ump and xhr. http://www.w3.org/2001/tag/2010/06/01-cross-domain.html, June 2010.

[31] A. Klein. Forging http request headers with flash. http://www.securityfocus.com/archive/1/441014/30/0/threaded, 2006.

[32] H. Leggatt. Gartner: Not long before mobile internet access exceeds access via pcs. http://www.bizreport.com/2010/01/gartner_not_long_before_mobile_internet_access_exceeds_acces.html, 2010.

[33] W. Maes, T. Heyman, L. Desmet, and W. Joosen. Browser protection against cross-site request forgery. November 2009.

[34] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. *LNCS*, 2009.

[35] Microsoft. Client-side cross-domain security. http://msdn.microsoft.com/library/cc709423.aspx, June 2008.

[36] Microsoft. Lack of flash support worries ipad publishers. http://www.msnbc.msn.com/id/35676003/ns/technology_and_science-tech_and_gadgets/t/lack-flash-support-worries-ipad-publishers/, February 2010.

[37] Microsoft. Silverlight: Making a service available across domain boundaries. http://msdn.microsoft.com/en-us/library/cc197955(v=vs.95).aspx, 2010.

[38] Mozilla. Http access control. https://developer.mozilla.org/en/http_access_control, 2010.

[39] OAuth. Initiative for open authentication. http://oauth.net/.

[40] OWASP. Owasp top 10 2007. http://www.owasp.org/index.php/Top_10_2007, 2007.

[41] OWASP. Open redirect. https://www.owasp.org/index.php/Open_redirect, Mei 2009.

[42] OWASP. Cross-site request forgery (csrf). http://www.owasp.org/index.php/Cross-Site_Request_Forgery, 2010.

[43] OWASP. Cross-site scripting (xss). http://www.owasp.org/index.php/XSS, 2010.

[44] OWASP. Owasp top 10 2010. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010.

[45] OWASÍP. Owasp csrfguard project. http://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project.

[46] D. Pranke. Chromium's support for cors and ump. http://lists.w3.org/Archives/Public/public-webapps/2010AprJun/0543.html, May 2010.

[47] P. D. Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. 2010.

[48] P. D. Ryck and L. Desmet. Csfire: Transparent client-side mitigation of malicious cross-domain requests. 2010.

[49] J. Samuel. Request policy 0.5.8. http://www.requestpolicy.com.

[50] T. Schreiber. Session riding: a widespread vulnerability in today's web applications. *SecureNet*, 2004.

[51] H. Shahriar and M. Zulkernine. Client-side detection of cross-site request forgery attacks. November 2010.

[52] C. Shifflet. Myspace csrf and xss worm (samy). http://shiflett.org/blog/2005/oct/myspace-csrf-and-xss-worm-samy, 2005.

[53] C. Shifflet. My amazon anniversary. http://shiflett.org/blog/2007/mar/my-amazon-anniversary, 2007.

[54] M. Stachowiak. Cors background. http://lists.w3.org/Archives/Public/public-webapps/2009OctDec/att-0468/CORS.pdf, November 2009.

[55] M. Stachowiak. [ump] request for last call. http://lists.w3.org/Archives/Public/public-webapps/2010AprJun/0043.html, April 2010.

[56] Symantec. Symantec reports first active attack on a dsl router. http://www.h-online.com/security/news/item/Symantec-reports-first-active-attack-on-a-DSL-router-735883.html, 2008.

[57] W3C. Access authentication. http://www.w3.org/Protocols/rfc2616/rfc2616-sec11.html.

[58] W3C. Comparison of cors and ump. http://www.w3.org/Security/wiki/Comparison_of_CORS_and_UMP.

[59] W3C. Cross-origin resource sharing. http://www.w3.org/TR/cors/.

[60] W3C. Rfc2616: Method definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

[61] W3C. Xmlhttprequest. http://www.w3.org/TR/XMLHttpRequest/.

[62] W3C. Http state management mechanism. http://www.w3.org/Protocols/rfc2109/rfc2109, 1997.

[63] W3C. Html5 specificatie. http://www.w3.org/TR/2010/WD-html5-20100624/, Juni 2010.

[64] W3C. Uniform messaging policy, level 1. http://www.w3.org/TR/UMP/, January 2010.

[65] W3C. Xmlhttprequest level 2. http://www.w3.org/TR/XMLHttpRequest2/, September 2010.

[66] M. Wiacek. Browser security handbook. http://code.google.com/p/browsersec/, 2011.

[67] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. *Princeton University*, 2008.

Fiche masterproef

Student: Maarten Lambert

Titel: Bescherming van mobiele browsers tegen cross-site request forgery

Engelse titel: Protecting mobile browsers against cross-site request forgery

UDC: 681.3

Korte inhoud:
In deze masterproef wordt de beveiliging tegen Cross-Site Request Forgery op mobiele platformen onderzocht.

De bijdrage van deze masterproef is drievoudig. Ten eerste wordt onderzoek geleverd naar de veiligheid van inter-domein communicatie. Dit onderzoek bestaat enerzijds uit een kritische bespreking van bestaande CSRF maatregelen. Anderzijds bestaat ze uit een overzicht van twee recent ontwikkelde specificaties die veilig inter-domein verkeer willen verwezenlijken, *Cross Origin Resource Sharing* (CORS), reeds ondersteund door de meeste browsers, en *Uniform Messaging Policy* (UMP), een systeem dat momenteel nog geen browser ondersteuning geniet.

De tweede bijdrage bestaat uit de ontwikkeling van een CSRF beschermingsmaatregel voor browsers van mobiele platformen in de vorm van een extensie aan de client-zijde. Deze vertrekt vanuit de werking van CsFire, een bestaande beveiliging voor Mozilla Firefox. Enerzijds wordt in deze masterproef een extensie voor Mozilla Fennec ontwikkeld, de mobiele versie van de populaire browser Mozilla Firefox. Dit is de eerste extensie voor Mozilla Fennec die rekening houdt met bestaande web 2.0 technieken en het gebruik hiervan zo minimaal mogelijk tracht te verstoren. Anderzijds brengt het onderzoek in deze masterproef ook een extensie voor de Android Browser voort, die standaard geïnstalleerd staat op alle Android toestellen. Dit is een primeur aangezien er nog geen CSRF maatregelen bekend zijn voor deze browser.

Een laatste bijdrage bestaat uit de combinatie van de eerste twee bijdragen. De bestaande extensies blokkeren de werking van *Cross Origin Resource Sharing* geheel of gedeeltelijk waardoor websites deze techniek niet meer kunnen toepassen. De extensies die uit de tweede bijdrage voortvloeiden, worden dan ook aangepast zodat ze een gepaste ondersteuning bieden voor CORS. De aanpassingen houden in dat de bestaande CORS werking lichtjes gewijzigd wordt, waardoor een veiligere variant op dit mechanisme ontstaat.

Thesis voorgedragen tot het behalen van de graad van Master in de ingenieurswetenschappen: computerwetenschappen

Promotoren: Dr. ir. L. Desmet
Prof. dr. ir. F. Piessens

Assessoren: Prof. dr. D. Clarke
Dr. P. Philippaerts

Begeleiders: P. De Ryck
Ir. S. Van Acker