KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Struikelblokken en kwetsbaarheden bij de adoptie van same-site cookies

Gertjan Franken

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotor:
Prof. dr. ir. Wouter Joosen

Assessoren:
Dr. S. Michiels
Dr. M. Vanhoef

Begeleider:
T. Van Goethem

Academiejaar 2016 – 2017

Voorwoord

Het schrijven van deze thesis was een machtige, maar voldoeninggevende taak. Ik kon gelukkig rekenen op de mensen rondom mij, zij stonden mij bij met raad, steun en afleiding. Voor deze mensen schrijf ik graag een dankwoordje. Eerst en vooral wil ik mijn begeleider Tom Van Goethem bedanken. Onder zijn goede begeleiding heb ik dit finale proefstuk kunnen voltooien. Tegelijkertijd heb ik ook veel bijgeleerd dankzij zijn expertise en duidelijk uitleg. Ook mijn promotor, prof. W. Joosen, wil ik bedanken om mij de kans te geven rond dit onderwerp een thesis te schrijven, maar ook voor de raad en tips die ik meegekregen heb tijdens onze contactmomenten. Ten slotte wil ik mijn familie en vrienden nog bedanken. Zonder hun steun en hoogkwalitatieve momentjes van ontspanning zou dit uiteraard heel wat moeilijker zijn geweest.

Gertjan Franken

Inhoudsopgave

Samenvatting

Cross-site aanvallen zoals *cross-site request forgery*, *cross-site script inclusion* en *cross-site timing attacks* vormen nog altijd een prominente dreiging op het web. Ondanks de verschillende verdedigingsmechanismen hiertegen, worden er nog altijd cruciale kwetsbaarheden vastgesteld bij populaire websites. Sinds kort ondersteunen de webbrowsers Google Chrome en Opera een nieuw verdedigingsmechanisme tegen deze aanvallen, namelijk de same-site cookie. Helaas gebeurt de adoptie van deze nieuwe vorm van bescherming traag, daar ook niet alle populaire webbrowser ondersteuning bieden. Hierdoor is er ook nog niet veel bekend over het gebruik, de werking en de effectiviteit van same-site cookies.

In deze thesis verrichten we verkennend onderzoek naar same-site cookies. We overlopen de eigenlijke werking van dit nieuwe concept en verhelderen enkele onduidelijkheden aanwezig in de Internet Draft van de same-site cookie. Dit doen we door zelf een gedetailleerde analyse uit te voeren met als doel het gedrag van same-site cookies in verschillende scenario's te achterhalen. Via dit verkennend onderzoek ontdekten we een bug en enkele inconsistenties, die we dan ook bespreken. We stellen ook een prototype voor dat de adoptie van de same-site cookie tracht te vergemakkelijken. Dit prototype is een uitbreidingsmodule voor webservers en kan voor verschillende soorten webservers geïmplementeerd worden. Ten slotte evalueren we dit prototype in combinatie met same-site cookies op basis van performantie en veiligheid tegen bestaande aanvallen. We tonen aan dat, ondanks de gevonden inconsistenties, same-site cookies een degelijk middel zijn om websites te voorzien van een extra laag van bescherming tegen cross-site aanvallen.

Lijst van figuren

Lijst van tabellen

Lijst van afkortingen en symbolen

Afkortingen

AES	Advanced Encryption Standard
API	Application Programming Interface
CDN	Content Delivery Network
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CSRF	Cross-Site Request Forgery
GCM	Galois/Counter Mode
HTTP	Hypertext Transfer Protocol
OWASP	Open Web Application Security Project
SOP	Same-Origin Policy
URL	Uniform Resource Locator
XHR	XmlHttpRequest
XSS	Cross-Site Scripting
XSSI	Cross-Site Script Inclusion

Hoofdstuk 1

Inleiding

Sinds de introductie van de oorspronkelijke HTTP cookie in 1994, is dit stukje webbrowserfunctionaliteit uitgegroeid tot een alomgebruikte webstandaard [1]. De mogelijkheid voor websites om kleine stukjes data op te slaan in de webbrowser maakte de weg vrij voor heel wat nieuwe toepassingen. De belangrijkste toepassing hiervan is misschien wel sessiebeheer via een zogenaamde session cookie. Een session cookie wordt gebruikt om een gebruikerssessie te onthouden nadat de gebruiker voor de eerste keer inlogt op een website. Op deze manier hoeft de gebruiker niet elke keer opnieuw in te loggen wanneer hij een nieuwe pagina van deze website opvraagt. Bij elke nieuwe request naar deze website wordt de session cookie namelijk automatisch meegestuurd zodat de website de ingelogde gebruiker kan identificeren.

Jammer genoeg heeft niet alleen de gebruiker baat bij toepassing van de session cookie, maar ook potentiële aanvallers. Aanvallers kunnen namelijk misbruik maken van de impliciete authenticatie die automatisch verricht wordt via de session cookie. Een bezoek aan de website van de aanvaller kan ervoor zorgen dat impliciet geauthenticeerde cross-site requests verstuurd worden naar de website waarop de gebruiker ingelogd is. Deze aanvallen worden ook wel cross-site aanvallen genoemd en vormen nog altijd een groot probleem op het web.

Een van de bekendste cross-site aanvallen is *Cross-Site Request Forgery* (CSRF). De aanvaller tracht hier impliciet geauthenticeerde requests te initiëren die een bepaalde actie autoriseren in naam van het slachtoffer op een bepaalde website. Deze acties kunnen gaan van het versturen van berichten tot het verrichten van een overschrijving op de website van een financiële instelling. De werkelijke impact van deze aanval werd gedemonstreerd door Zeller et al., zij toonden aan dat grote websites van onder andere Youtube en ING Direct kwetsbaar zijn voor CSRF-aanvallen [40].

Naast CSRF gebruikt *Cross-Site Script Inclusion* (XSSI) ook session cookies, maar deze keer om gevoelige informatie te stelen. De aanvaller omzeilt de Same-Origin Policy om de gebruikersspecifieke informatie die door een impliciet geauthenticeerde request is opgehaald, te lekken. Er zijn verschillende varianten van deze aanval die gericht zijn op verschillende vormen van informatie. Zo slaagden Lekies et al. er onder andere in om het e-mailadres en de kalenderafspraken van gebruikers van een bekende e-mail provider bemachtigen [18]. Het doelwit van deze aanval was een

dynamisch JavaScript-bestand gegenereerd op basis van de session cookie.

Timing attacks kunnen ook toegepast worden als cross-site aanval. De cross-site timing attack werd geïntroduceerd door Bortz en Boneh [4]. Door middel van de impliciet geauthenticeerde request wordt dan niet getracht informatie te stelen, maar te construeren uit de tijd die nodig is om een bepaalde resource te downloaden. Deze tijdsmeting zou dan bepaalde informatie over de gebruiker kunnen onthullen, zoals het feit of de gebruiker ingelogd is op een bepaalde website. De browser-based timing attack, geïntroduceerd door Van Goethem et al., is een variant hierop die gebruik maakt van side-channel lekken in de webbrowser [10]. Cross-site resources worden hier nog steeds opgevraagd, maar de tijdsmeting wordt enkel toegepast op het verwerken van de resource door de webbrowser. Dit maakt de aanval veel nauwkeuriger. Van Goethem et al. demonstreerden bijvoorbeeld hoe achterhaald kan worden tot welke doelgroep een Facebook-gebruiker behoort door middel van een browser-based timing attack.

Voor deze cross-site aanvallen bestaan er al verschillende beschermingsmechanismen en -maatregelen, zowel gericht op specifieke aanvallen als op cross-site aanvallen in het algemeen. Toch worden er nog steeds veel kwetsbaarheden voor deze aanvallen ontdekt. Zo blijft CSRF op de achtste plaats staan in de *release candidate* van de OWASP Top Ten Project van 2017 [24]. Daarnaast werden er kwetsbaarheden voor de andere aanvallen ontdekt in recente papers [18, 10]

De same-site cookie is voorgesteld als een nieuw *in-depth* beschermingsmechanisme tegen bovenstaande aanvallen [36]. Een same-site cookie is een gewone cookie die websites toelaat om, via een extra attribuut, aan te geven met welke cross-site requests deze cookie mag meegestuurd worden. Op deze manier kan de website ervoor zorgen dat bepaalde cross-site requests niet meer geauthenticeerd worden door de session cookie. Dit zal het moeilijker maken voor aanvallers om bovenstaande aanvallen uit te kunnen voeren. Het gaat hier echter wel nog om een zeer nieuwe functionaliteit, de same-site cookie wordt alleen nog maar ondersteund door Google Chrome en Opera sinds 2016. Dit zorgt er mede voor dat de adoptie door websites zeer traag gebeurt. Daarnaast kan de adoptie ook voor struikelblokken zorgen omdat het integreren van same-site cookies waarschijnlijk enkele aanpassingen zal vereisen in de reeds gebruikte cookiestrategie. We zijn tijdens dit onderzoek nog maar één website tegengekomen die same-site cookies gebruikt. Dit is de website van Dropbox. In hun blog lieten ze eerder al weten dat ze overwogen om same-site cookies te gaan gebruiken [6]. Voor de rest hebben we ook nog geen wetenschappelijk onderzoek gevonden omtrent same-site cookies.

In deze thesis onderzoeken we het eigenlijke functionele gedrag van de same-site cookie. Omdat we geen onderzoek vonden over same-site cookies en de same-site cookie Internet Draft bepaalde vragen onbeantwoord liet, zijn we gestart met een analyse van same-site cookies in cross-site requests. Tegelijkertijd toetsen we ook de werking beschreven in deze Internet Draft met de implementatie van de webbrowsers. De inconsistenties en bug die we tegengekomen zijn, worden in deze thesis besproken. Om de adoptie van de same-site cookie te bevorderen, hebben we een prototype ontwikkeld. Dit prototype kan server-side worden gebruikt ter vergemakkelijking van de integratie van same-site cookies op een website. De bedoeling hiervan is dat

same-site cookies gebruikt kunnen worden zonder dat de bestaande functionaliteit van de website hieronder moet lijden. Uiteindelijk evalueren we dit prototype door het toe te passen op een testomgeving. We meten hierbij de extra CPU-tijd die de webserver zal nodig hebben om dit prototype te gebruiken en beoordelen de veiligheid met betrekking tot bestaande aanvallen.

Onze belangrijkste bijdragen zijn de volgende:

- We analyseren en toetsen de werking van same-site cookies voor cross-site requests die op verschillende wijzen geïnitieerd zijn.
- We beschrijven een server-side prototype in de vorm van een Apache-module die de adoptie van same-site cookies tracht te vergemakkelijken.
- We evalueren dit prototype in combinatie met de functionaliteit van de same-site cookie op basis van performantie en bescherming tegen bestaande aanvallen.

In het volgende hoofdstuk geven we achtergrondinformatie over de aanvallen waarop de bescherming door same-site cookies gericht is. We bespreken hier ook bestaande verdedigingsmechanismen en gerelateerd werk omtrent deze aanvallen. In hoofdstuk 3 leggen we in detail uit wat de same-site cookie nu eigenlijk is en hoe deze gebruikt kan worden. Dit hoofdstuk behandelt ook het verkennend onderzoek en de resultaten die hieruit zijn voortgekomen. De werking van het ontwikkelde prototype ter vergemakkelijking van de adoptie van same-site cookies bespreken we in hoofdstuk 4. Dit prototype evalueren we in combinatie met de same-site cookie in hoofdstuk 5. Ten slotte sluiten we af met een discussie in hoofdstuk 6 en een besluit in hoofdstuk 7.

Hoofdstuk 2

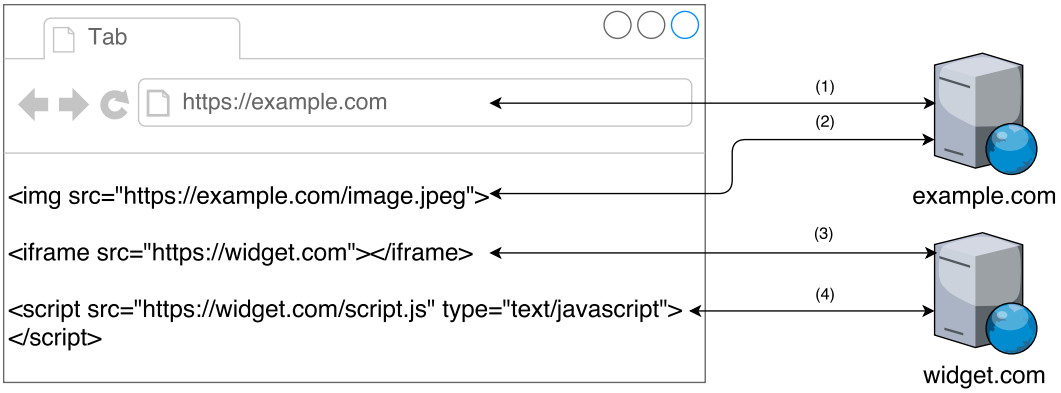
Achtergrond

In dit hoofdstuk schetsen we de context in welke de same-site cookie werd geïntroduceerd. Eerst wordt er uitgelegd wat cross-site requests, cookies en de Same-Origin Policy zijn (sectie 2.1). De meest prominente aanvallen die gebruik maken van cross-site requests worden daarna beschreven (sectie 2.2). We sluiten dit hoofdstuk af met een bespreking over gerelateerd werk betreffende deze aanvallen en ontworpen verdedigingsmechanismen (sectie 2.3).

2.1 Cross-site requests, cookies en de Same-Origin Policy

Webpagina's kunnen resources invoegen door te refereren naar de URL van die resource. Dit gebeurt bijvoorbeeld door middel van de <img>-tag om afbeeldingen in te voegen op een HTML-pagina. De webbrowser zal bij het bezoeken van deze webpagina de afbeelding ophalen door de gespecifieerde URL te raadplegen. Er wordt hiervoor een request gestuurd naar die specifieke URL door de webbrowser. Dit gebeurt automatisch voor alle ingevoegde resources. Neem bijvoorbeeld het scenario afgebeeld in figuur 2.1. De gebruiker heeft het adres `https://example.com` ingegeven in de adresbalk en als gevolg maakt de webbrowser een request (1). Als response ontvangt de webbrowser de HTML-pagina afgebeeld in het webbrowservenster. Deze HTML-pagina bevat drie referenties naar resources. Voor de <img>-tag zal de browser een request (2) versturen naar de webserver met domeinnaam `example.com`. Dit is een same-site request omdat er gerefereerd wordt vanaf een webpagina afkomstig van hetzelfde domein (`example.com`). De overige resources bevinden zich daarentegen op het domein `widget.com`. Ook hiervoor verstuurt de webbrowser requests (3 en 4) om deze resources in te voegen. Dit zijn cross-site requests omdat deze resources zich bevinden op een ander domein (`widget.com`).

Zowel same-site als cross-site requests kunnen een vorm van authenticatie dragen. Deze authenticatie kan gebeuren op verschillende manieren, maar in deze thesis concentreren we ons op authenticatie door middel van HTTP cookies [1]. Een cookie is een klein stukje data dat opgeslagen wordt in de webbrowser. Dit gebeurt in naam van een website die de webbrowser heeft bezocht. Door middel van de response-header



Figuur 2.1: Voorbeelden van same-site en cross-site requests.

`Set-Cookie` definieert de website de cookie die het wilt opslaan. De webbrowser zal nu elke keer deze cookie meesturen in de `Cookie`-header van een request naar deze website. Deze cookies worden vaak gebruikt om de gebruiker van een website te identificeren, zodat deze niet elke keer opnieuw hoeft in te loggen. Dit worden ook wel session cookies genoemd. Omdat session cookies automatisch worden meegestuurd en als gevolg de request geauthenticeerd is in naam van een bepaalde gebruiker, noemen we dit impliciete authenticatie. In het voorbeeld van figuur 2.1 zal er dus in alle vier de requests impliciete authenticatie aanwezig zijn in de vorm van een cookie, indien deze is ingesteld door de desbetreffende website.

Het zou onveilig zijn mochten scripts op de oorspronkelijke webpagina vrije toegang hebben tot alle ingevoegde resources. Deze resources, die mogelijk verkregen werden via een impliciet geauthenticeerde request, kunnen dan via deze scripts gelekt worden. Dit wordt gelukkig voorkomen door de *Same-Origin Policy* (SOP) [2], een policy die wordt afgedwongen door de webbrowser. De SOP schrijft voor dat scripts geen toegang hebben tot resources die niet afkomstig zijn van dezelfde origin. De origin van een script is afhankelijk van waar het script wordt uitgevoerd, niet van waar het wordt geïmporteerd. Een origin staat voor de combinatie van protocol, domein en poort in de vorm `protocol://domein:poort`. We kunnen de SOP toepassen op het voorbeeld uit figuur 2.1. Het script dat ingeladen wordt vanuit `https://widget.com/script.js`, wordt uitgevoerd in de context van `https://example.com`. In combinatie met de poort, is dit is de origin van het script. Omdat de afbeelding afkomstig is van dezelfde origin, laat de SOP toe dat het script toegang heeft tot de inhoud van deze afbeelding. De SOP laat echter niet toe dat het script toegang heeft tot de iframe, omdat de origins van deze twee elementen verschillen.

Omdat de SOP soms te streng is om een bepaalde functionaliteit voor een webapplicatie te kunnen realiseren, bestaan er methoden om het SOP-beleid te verzachten. Zo kan de toegang tot resources afkomstig van subdomeinen worden toegestaan door `document.domain` aan te passen. Via `document.domain` kunnen subdomeinen namelijk de domeinnaam van een superdomein toegewezen krijgen, om

zo hun origins overeen te laten komen. Daarnaast is het ook mogelijk om JSON-bestanden met een andere origin in te lezen door middel van een script. Hiervoor wordt *JSON with Padding* (JSONP) gebruikt, waarbij de JSON-data wordt ingesloten in een script. Dit kan echter misbruikt worden door een aanvaller met *Cross-Site Script Inclusion* (XSSI). JSONP en XSSI bespreken we uitgebreid in sectie 2.2.3. Een veiligere oplossing is *Cross-Origin Resource Sharing* (CORS) [17]. Door middel van de `Origin`- en `Access-Control-Allow-Origin`-header wordt de toegang tot cross-origin resources toegelaten.

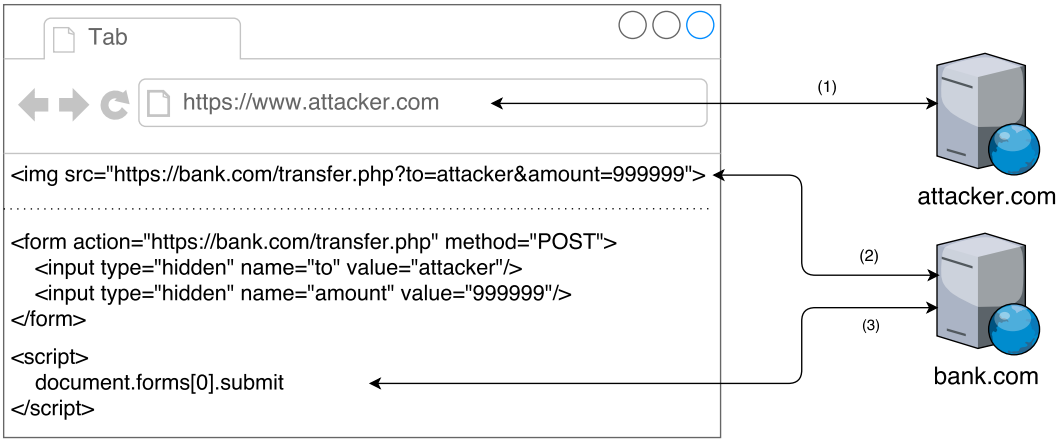
2.2 Bestaande aanvallen op basis van cross-site requests

De meest voorkomende aanvallen die gebruik (kunnen) maken van cross-site requests zijn *Cross-Site Scripting* (XSS) en *Cross-Site Request Forgery* (CSRF). Hoewel beide aanvallen gedaald zijn in de OWASP Top Ten Project van 2013 [23] ten opzichte van de 2010-editie, komen ze nog altijd veel voor. In de voorlopige *release candidate* van 2017 behouden ze hun plaats [24]. Minder bekende aanvallen op basis van cross-site requests staan bekend onder de namen *Cross-Site Script Inclusion* (XSSI) en cross-site timing attacks. Deze vier aanvallen bespreken we kort in deze sectie.

2.2.1 Cross-Site Request Forgery

Een CSRF-aanval kan geïnitieerd worden wanneer een gebruiker de website van een aanvaller bezoekt. De bezochte pagina kan ervoor zorgen dat de webbrowser van de gebruiker een cross-site request verstuurt naar een bepaalde webserver, zonder dat de gebruiker hier weet van heeft. Indien de gebruiker geauthenticeerd is op deze website door middel van een session cookie, wordt deze cookie automatisch meegestuurd. De request is dus impliciet geauthenticeerd. Als gevolg zal de webserver dit geauthenticeerde verzoek behandelen alsof de gebruiker dit intentioneel verstuurde. Dit kan uiteindelijk leiden tot ernstige schade voor de gebruiker, zeker wanneer het gaat om de webserver van een overheids- of financiële instelling. Dergelijke kwetsbaarheden werden al vastgesteld bij belangrijke websites van onder meer YouTube, ING Direct, The New York Times en MetaFilter [40].

Er bestaan verschillende manieren om een cross-site request te versturen vanaf een HTML-pagina. Voor CSRF kunnen bijvoorbeeld <iframe>- en <img>-tags gebruikt worden. De browser verzendt dan automatisch een verzoek naar de webserver om de inhoud van de iframe of afbeelding op te halen. Daarnaast is het ook mogelijk om via clickjacking ervoor te zorgen dat de gebruiker op een hyperlink klikt waardoor er eveneens een verzoek verstuurd wordt. Bij elk van deze methoden zal de webbrowser een GET-request versturen naar de webserver. De HTTP-parameters in deze request kunnen dan gebruikt worden om een bepaalde actie uit te voeren. Er kan ook een POST-request worden verstuurd door de webbrowser, bijvoorbeeld door gebruik van de <form>-tag. De HTML-form gedefinieerd door de aanvaller kan zelfs automatisch ingediend worden met behulp van JavaScript. Figuur 2.2 toont een voorbeeld van



Figuur 2.2: Twee voorbeelden van een CSRF-aanval op `bank.com`.

hoe een CSRF-aanval voor elk van deze twee methoden kan uitgevoerd worden. De gebruiker komt op een of andere manier op de website van de aanvaller terecht, hier `www.attacker.com`, en het HTML-document wordt opgevraagd (1). Voor de eerste methode wordt een automatische request (2) verstuurd omwille van een <img>-tag. De bron-URL van deze <img>-tag wordt echter gebruikt om een overschrijving uit te voeren naar de rekening van de aanvaller op `www.bank.com`. Voor de andere methode wordt er ook weer een automatisch request (3) verstuurd, maar deze keer omwille van de indiening van een HTML-form via JavaScript. Hier wordt dezelfde overschrijving uitgevoerd, maar in plaats van een GET-request wordt er een POST-request verstuurd.

GET- en POST-requests kunnen ook verstuurd worden door middel van *XMLHttpRequest* (XHR) [38]. Deze aanval kan dus op een gelijkaardige manier als in het voorbeeld uitgevoerd worden met XHR, zelfs met HEAD-requests. Ook andere HTTP-requestmethoden kunnen door XHR worden gebruikt zoals PUT en DELETE, maar niet ten behoeve van een CSRF-aanval. Hier zal de eigenlijke request voorafgegaan moeten worden door een *preflight*-request. Dit wordt gedicteerd door de *Cross-Origin Resource Sharing*-standaard (CORS) [17, 20]. Zo een *preflight*-request vraagt in essentie of de webbrowser toestemming heeft om een bepaalde requestmethode te versturen. Dankzij deze standaard kan een website bepalen welke requestmethoden toegelaten zijn voor welke domeinen.

We bespreken kort enkele belangrijke maatregelen die genomen kunnen worden tegen CSRF-aanvallen.

CSFR-tokens

Het gebruik van CSRF-tokens is de meest effectieve verdediging tegen CSRF-aanvallen. Dit kan toegepast worden in de vorm van het *Synchronizer Token Pattern*. Voor elke gebruikerssessie wordt er een unieke en willekeurige waarde gegenereerd. Dit is de CSRF-token en de webserver slaat deze op voor elke gebruikerssessie. Deze

token wordt in elke HTML-form meegestuurd voor een specifieke gebruikerssessie als verborgen waarde. Dankzij de SOP heeft een aanvaller geen toegang tot deze token. Bij het indienen van de HTML-form door de gebruiker, zal de token ook weer mee terug worden meegestuurd naar de webserver. Uiteindelijk kan de webserver valideren of deze token overeenkomt met de opgeslagen token. Indien dit niet het geval is, kan men vermoeden dat het om een CSRF-aanval gaat en de indiening negeren. In het voorbeeld van figuur 2.2 zou de webserver de indiening van de HTML-form niet accepteren omdat er geen CSRF-token wordt meegestuurd. HTML-forms die daarentegen afkomstig zijn van `bank.com`, zullen deze token wel bevatten.

De *Double Submit Cookie* is een vergelijkbare oplossing. In plaats van de CSRF-token server-side op te slaan, wordt deze opgeslagen in de vorm van een cookie. De token wordt dus twee keer verstuurd naar de webbrowser, één keer in een verborgen veld in de HTML-form en één keer in de vorm van een cookie. Dankzij de SOP hebben enkel scripts afkomstig van dezelfde origin toegang tot deze cookie. De webserver valideert dan of de token van de ingediende HTML-form overeen komt met de token in de cookie.

Het *Encrypted Token Pattern* is een andere vergelijkbare oplossing waarbij er ook geen extra token server-side moet worden opgeslagen. De geëncrypteerde session id wordt hier gebruikt als token. Bij het terug ontvangen van de token wordt deze gedecrypteerd en vergeleken met de session id ter validatie.

De Referer-, Origin- en aangepaste HTTP-header

Aan de hand van de `Referer`-header kan een webserver zien van waar een request werd geïnitieerd. Indien een gebruiker op een hyperlink klikt - en bijgevolg naar de webpagina van deze link surft - zal de webbrowser met de `Referer`-header aangeven op welke webpagina deze hyperlink zich bevond. Dit kan gebruikt worden door de webserver om te onderzoeken vanwaar zijn bezoekers afkomstig zijn, maar ook om te controleren of een request afkomstig is van een vertrouwde bron. De Referer-header heeft echter wel een aantal nadelen. Zo is de privacy van de gebruiker een heikel punt. Niet iedere gebruiker vindt het aanvaardbaar dat deze informatie zomaar wordt vrijgegeven. Daarbij kan de URL in deze header bijvoorbeeld zoektermen bevatten indien de gebruiker op een link klikte bij gebruik van een zoekmachine. Bovendien wordt deze header vaak gespooft of gestript[3], al dan niet net om wille van het privacyprobleem. Hierdoor kunnen sommige websites niet anders dan requests met ontbrekende `Referer`-header te accepteren om de functionaliteit voor alle gebruikers te garanderen.

In januari van 2017 is er een *candidate recommendation* gepubliceerd bij het *World Wide Web Consortium* (W3C) voor de invoering van een `Referrer-Policy`-header [8]. Deze response-header geeft administrators de mogelijkheid om te specifiëren welke informatie er in de `Referer`-header moet worden vermeld voor requests die origineren vanaf hun website. De specificatie heeft het over negen voorgedefinieerde waarden die aan deze header kunnen worden gegeven (inclusief de lege waarde). Elk van deze waarden bepaalt hoe de webbrowser de `Referer`-header zal instellen bij een request afkomstig van deze website. Privacy en beveiliging zijn de voornaamste

redenen voor dit voorstel. Websites beschikken op deze manier over een fijnere controle wat betreft de gelekte informatie via de `Referer`-header. Enkele interessante voorbeelden van `Referrer-Policy`-instellingen zijn `no-referrer` en `same-origin`. Bij deze eerste wordt er nooit een `Referer`-header ingesteld door de webbrowser. Bij de laatste zal de webbrowser enkel de `Referer`-header instellen wanneer het gaat om een request naar dezelfde origin als de oorspronkelijke webpagina. Indien de `Referrer-Policy`-header er werkelijk voor zou kunnen zorgen dat websites deze gaan gebruiken om de privacy van hun gebruikers te beschermen, zouden gebruikers in mindere mate geneigd kunnen zijn de `Referer`-header strippen.

Als alternatief voor de Referer-header werd de `Origin`-header voorgesteld als bruikbare header ter verdediging tegen CSRF [3]. Deze header is onderdeel van de CORS-standaard [17]. Eerder dan de complete URL bevat deze header de origin (protocol, domein en poort) waarvan de request afkomstig is. Dit vermindert de bezorgdheid rond privacy, waardoor gebruikers deze header ook minder snel moedwillig zouden spoofen of strippen. De header moet meegestuurd worden met elke POST-request, maar dit hoeft niet voor GET-requests. GET-requests mogen dan wel zeker geen invloed hebben op de staat van de webserver, dit mag enkel nog via POST-requests. Hoewel dit eigenlijk de standaard is, laten veel websites nog steeds toe dat staatswijzigende requests via GET gebeuren. De webserver kan dan elke request afwijzen waarvan de `Origin`-header een ongewenste waarde heeft. Als gevolg van inconsistente implementaties van de `Origin`-header tussen de verschillende webbrowsers, is het echter moeilijk voor websites om deze header te gebruiken ter verdediging tegen CSRF. Zo zal Firefox nooit een `Origin`-header meesturen in een same-origin request, terwijl Google Chrome en Safari dit wel doen. Indien een webserver een request zonder `Origin`-header ontvangt, kan deze afkomstig zijn van een Firefox-webbrowser of van een oude webbrowser die de `Origin`-header nog niet ondersteunt. Als men ervan uit gaat dat het ontbreken van de `Origin`-header duidt op een same-origin request, dan zijn oudere webbrowsers nog steeds kwetsbaar voor een CSRF-aanval.

CSRF-aanvallen kunnen ook voorkomen worden met behulp van aangepaste HTTP-headers [3]. Enkel same-site requests kunnen aangepaste HTTP-headers dragen door het gebruik van XHR, zonder expliciete toestemming via de `Access-Control-Allow-Headers`-header [17]. De webserver kan afdwingen dat elke legitieme toestandswijzigende request een bepaalde aangepaste HTTP-header moet dragen. Met andere woorden zal dus elke toestandswijzigende request via XHR moeten gebeuren. De benodigde aangepaste HTTP-header zorgt ervoor dat cross-site requests hiervoor niet in aanmerking komen (zonder expliciete `Access-Control-Allow-Headers`-header) en op deze manier geen CSRF-aanval uitgevoerd kan worden.

Client-side verdediging

Sinds de allereerste client-side verdedigingsapplicatie tegen CSRF-aanvallen (RequestRodeo [16]) zijn er nu veel meer initiatieven die dit voorbeeld volgen. Vaak worden deze applicaties aangeboden in de vorm van webbrowserextensies die ge-

makkelijk te installeren zijn. Deze extensies hebben toegang tot de API van de webbrowser en kunnen zo HTTP-requests en -responses controleren en manipuleren. Daarnaast zijn ze meestal ook gemakkelijk te configureren door de gebruiker zelf én worden ze automatisch geüpdatet. De gebruiker heeft dus zelf de mogelijkheid om in te staan voor zijn eigen veiligheid en hoeft daarvoor niet blind te vertrouwen op de aanbieder van een service. Het meerendeel van deze extensies focust zich op het onderscheppen van cross-site requests. Dit gebeurt dan op basis van policies die ingesteld worden door de gebruiker, door de uitgever of een combinatie hiervan. Enkele voorbeelden hiervan zijn CsFire[1] en uMatrix[2] voor Google Chrome en Mozilla Firefox, en RequestPolicy[3] voor Mozilla Firefox. Er zijn ook extensies die voorkomen dat kwaadwillige client-side scripts worden uitgevoerd door de webbrowser. Een bekend voorbeeld hiervan is NoScript[4] voor Mozilla Firefox. Dit maakt het moeilijker voor de aanvaller om automatisch cross-site requests te veroorzaken. Denk maar aan hoe een form automatisch kan worden ingediend of hoe GET-, POST- en HEAD-requests verstuurd kunnen worden door middel van JavaScript. In sectie 2.3 over gerelateerd werk, bespreken we de allereerste client-side verdediging RequestRodeo en de meer recente oplossing CsFire.

2.2.2 Cross-Site Scripting

Een XSS-aanval maakt niet noodzakelijk gebruik van cross-site requests. Omdat het echter gebruikt kan worden als luik naar een cross-site aanval, bespreken we hier toch kort deze aanval.

Bij een XSS-aanval injecteert de aanvaller een stukje client-side code in de doelwebsite. Dit kan hij bijvoorbeeld doen op een pagina die gebruikers toelaat om een reactie te plaatsen. Deze reactie is dan zichtbaar voor andere gebruikers wanneer ze de pagina in kwestie bezoeken. De inhoud van de reactie kan echter een script bevatten dat door de webbrowser van deze gebruikers automatisch wordt uitgevoerd. Dit script zou onder andere een cross-site aanval kunnen initiëren. Op deze manier hoeft de aanvaller dus niet noodzakelijk zijn eigen website te gebruiken voor een cross-site aanval.

Bescherming tegen XSS-aanvallen ligt buiten de scope van deze thesis. Hoewel de industriestandaard nog altijd berust op het valideren van gebruikersinvoer, wordt er steeds meer gebruik gemaakt van andere preventieve maatregelen, maar ook van detectiemechanismen. Onderzoek focust zich vooral op het voorkomen van aanvallen en het detecteren van zowel kwetsbaarheden als aanvallen [15].

2.2.3 Cross-Site Script Inclusion

De Same-Origin Policy belet scripts die uitgevoerd worden op de ene webpagina de toegang tot data afkomstig van een andere webpagina indien deze webpagina's

[1] https://distrinet.cs.kuleuven.be/software/CsFire
[2] https://github.com/gorhill/uMatrix
[3] https://www.requestpolicy.com/
[4] https://noscript.net/

niet dezelfde origin hebben. Dit wordt echter omzeild door gebruik van *JSON with Padding* (JSONP). Hier wordt de eigenlijke data in JSON-formaat ingesloten door een functie-oproep in JavaScript. Dit kan dus eigenlijk beschouwd worden als een script dat bestaat uit een functie-oproep met de eigenlijke data als parameter. Als men dan op de oorspronkelijke pagina de functie van de functie-oproep definieert en via de <script>-tag het script inlaadt, dan heeft men alsnog toegang tot de data. We beschouwen even een voorbeeld. Listing 2.1 toont het script dat geladen wordt vanaf een andere webpagina. De callback-functie wordt meegegeven in een argument bij het verzoek. Listing 2.2 toont het scriptgedeelte dat zich bevindt op de oorspronkelijke HTML-pagina. De leak-functie zal hier voor een pop-up zorgen waarin de opgehaalde data weergegeven wordt.

Listing 2.1: Het script dat toegankelijk is op de pagina https://example.com/data?callback=leak.

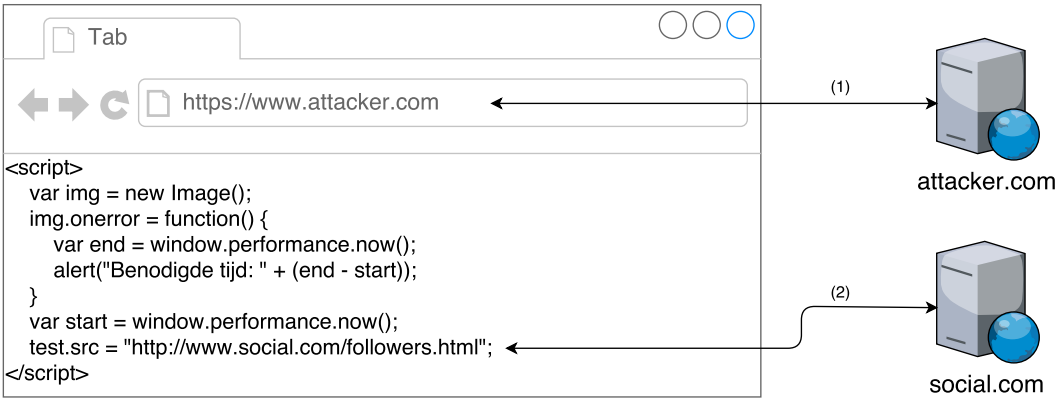
```
leak( { /* JSON data */ } );
```

Listing 2.2: De declaratie van de insluitende functie en verzoek tot ophalen van het script.

```
<script>
    function leak(data) { alert(JSON.stringify(data)); }
</script>
<script scr="https://example.com/data?callback=leak" type="text/javascript" >
</script>
```

Dit voorbeeld toont aan dat de data afkomstig van een andere website toch gebruikt kan worden via JSONP. Hier kan de aanvaller echter ook gebruik van maken. Op dezelfde manier als in het voorgaande voorbeeld kan de aanvaller op zijn eigen website de code uit listing 2.2 schrijven. Hij kan de leak-functie zo definiëren dat deze de gelekte data verzendt naar een webserver van de aanvaller. Dit stelt een XSSI-aanval voor met statisch JavaScript-bestand dat gevoelige informatie bevat. Wanneer een geauthenticeerd gebruiker de website van de aanvaller bezoekt, zal er een cross-site request worden verstuurd door de browser om het script van `example.com` op te halen. De impliciete authenticatie zal hierbij worden meegestuurd waardoor de webserver toegang verschaft tot het script met gevoelige informatie. Als gevolg kan de aanvaller deze informatie lekken. Naast statische JavaScript-bestanden met JSON-data, zijn dynamisch gegenereerde JavaScript-bestanden ook kwetsbaar voor een XSSI-aanval [18].

XSSI-aanvallen worden mogelijk gemaakt door het opslaan van gegevens in vormen die niet alleen toegankelijk zijn voor de legitieme instanties, maar ook voor aanvallers. JSONP maakt het mogelijk om gegevens te delen met vertrouwde domeinen, maar sluit andere domeinen niet uit. Daarom is het geen goed idee om vertrouwde informatie te delen door middel van JSONP. Het is aangeraden om in de plaats gebruik te maken van *Cross-Origin Resource Sharing* (CORS) [17]. Deze gegevens kunnen dan nog altijd in de vorm van JSON-bestanden toegankelijk worden



FIGUUR 2.3: Voorbeeld van een timing attack op `social.com`.

gemaakt voor vertrouwde domeinen. Omdat gegevens aanwezig in JavaScript-code ook gelekt kunnen worden, kunnen deze beter opgeslagen worden in bestanden die niet uitvoerbaar zijn als script. Zowel header-checking als client-side bescherming zoals besproken in sectie 2.2.1 kunnen hier ook gebruikt worden ter verdediging.

2.2.4 Timing attacks

Net zoals de hiervoor besproken aanvallen steunt de cross-site timing attack op impliciet geauthenticeerde cross-site requests. Het verschil is echter dat de aanvaller in dit geval niet de intentie heeft om de staat van de webserver te beïnvloeden of om de data op de webserver te bemachtigen. Hij observeert daarentegen side-channel lekken om zo informatie over een slachtoffer te vergaren. Stel dat een bepaalde website gebruikers toelaat zich te registreren. Ingelogde gebruikers krijgen toegang tot bepaalde inhoud van deze website in de vorm van bijvoorbeeld artikels en foto's. Een niet-ingelogde gebruiker krijgt een klein welkomsbericht met de suggestie zich te registreren. Het verschil in de aangeboden webpagina voor de ingelogde gebruiker ten opzichte van de niet-ingelogde gebruiker is significant. Hier kan de aanvaller gebruik van maken om te bepalen of zijn slachtoffer lid is van deze website of niet. Op zijn eigen website construeert de aanvaller een pagina die de webbrowser van het slachtoffer een cross-site request laat verzenden naar de website in kwestie. Daarnaast meet hij met behulp van een script de tijd die nodig is voor de webbrowser om die cross-site request te behandelen. Bij een lange laadtijd kan de aanvaller ervan uitgaan dat het slachtoffer lid is en ingelogd is op de website omdat er in dit geval een cookie werd meegestuurd die de persoon authenticeerde. De inhoud van artikels en foto's werd dus geladen. Bij een korte laadtijd, de tijd die nodig was om het korte welkomsbericht te laden, gaat hij uit van het tegenovergestelde.

Figuur 2.3 toont een voorbeeld van zo een timing attack. Deze aanval is gebaseerd op het aantal volgers dat een gebruiker van de website `social.com` zou hebben. Als een gebruiker van `social.com` de website van de aanvaller bezoekt, zal er een tijdsmeting uitgevoerd worden voor het ophalen van de pagina die alle volgers oplijst.

In de praktijk is het echter niet zo vanzelfsprekend om op basis van tijdsmetingen dergelijke conclusies te trekken. De laadtijd van een pagina wordt immers beïnvloed door externe factoren zoals fluctuaties in de verbindingssnelheid van het netwerk. Meerdere meetingen en de toepassing van statistiek kunnen dan worden gebruikt om een conclusie meer gewicht te geven. Een andere manier om dit probleem te omzeilen is de recent ontdekte browser-based timing attack [10]. Deze variant meet de tijd die een browser nodig heeft om elementen op een webpagina te verwerken. De downloadtijd wordt hier niet bijgeteld waardoor de tijdsmeting onafhankelijk is van fluctuaties in de verbindingssnelheid.

Als de webserver voor iedere request evenveel tijd nodig zou hebben, dan zou er geen informatie op basis van responstijd gelekt kunnen worden. Het is echter niet vanzelfsprekend om op elke request met een constante tijd te antwoorden. Daarnaast zou dit enkel weerstand bieden tegen de cross-site timing attack, maar niet tegen de browser-based timing attack. Deze laatste gebruikt immers lekken die enkel afhangen van de webbrowser. Net zoals besproken in sectie 2.2.1 kunnen header-checking en client-side bescherming hier ook gebruikt worden ter verdediging. In sectie 2.3 bespreken we enkele defensieve maatregelen die besproken zijn in de papers van deze aanvallen.

Size-exposing attacks

Hoewel deze aanval niet vernoemd wordt in de same-site cookie Internet Draft, is de cross-site size-exposing attack ook een aanval die we bespreken in deze thesis. Deze aanval gelijkt heel erg op de browser-based timing attack, er wordt hier ook gebruik gemaakt van informatie die gelekt wordt door de webbrowser. De grootte van een resource is de informatie die gelekt wordt via APIs die beschikbaar worden gesteld door de webbrowser. Deze resources kunnen worden opgehaald door middel van impliciet geauthenticeerde cross-site requests. Men heeft al aangetoond dat aan de hand van de grootte van een opgehaalde resources gevoelige informatie over de identiteit van een slachtoffer achterhaald kan worden [11].

2.3 Gerelateerd werk

Naar onze kennis is er nog geen wetenschappelijk werk verschenen over same-site cookies. In dat opzicht denken we ook dat het prototype beschreven in deze thesis het eerste is dat voorgesteld wordt ter vergemakkelijking van de adoptie van same-site cookies. Er zijn echter wel verschillende onderzoeken gepubliceerd over aanvallen die nauw verband houden met de same-site cookie. Werken over deze aanvallen, en hun voorgestelde verdedigingsmaatregelen, bespreken we in deze sectie.

2.3.1 Cross-site aanvallen

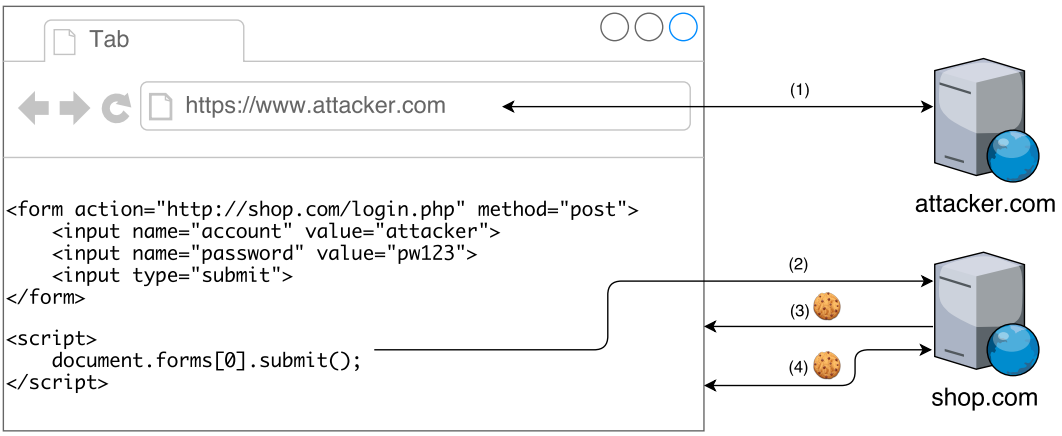
De same-site cookie zou een geschikte extra laag van verdediging kunnen bieden tegen cross-site aanvallen. We bespreken hier belangrijke onderzoeken omtrent deze aanvallen.

Login CSRF

Barth et al. bespreken Login CSRF, een variant van de bekende CSRF-aanval [3]. Hierbij tracht de aanvaller het slachtoffer te laten inloggen met een bepaald account, op een website die het slachtoffer gebruikt. Dit account is aangemaakt door de aanvaller, hij beschikt dus over de vereiste inloggegevens. We overlopen deze aanval aan de hand van het voorbeeld in figuur 2.4. Het slachtoffer is een gebruiker van de website `shop.com` en om een of andere reden bezoekt hij de website van de aanvaller. Als gevolg wordt deze pagina wordt ingeladen door de webbrowser (1). Met de inloggegevens van een zelf aangemaakt account laat de aanvaller het slachtoffer via een gewone CSRF-aanval inloggen (2). Wanneer het slachtoffer is ingelogd op dit account, zal er een session cookie ingesteld worden zoals `shop.com` voor iedere gebruiker doet (3). Elke actie die het slachtoffer nu uitvoert, gebeurt nu in naam van het account van de aanvaller door middel van impliciet geauthenticeerde requests (4). Afhankelijk van de website, bestaat de kans dat het slachtoffer niet door heeft dat het niet zijn eigen account is waarop hij is ingelogd. Als gevolg zal de gebruiker de website gebruiken zoals hij normaal zou doen; aanvullen van persoonlijke informatie, versturen van berichten, gebruiken van de zoekfunctie, etc. Omdat de aanvaller beschikt over de inloggegevens van dit account, kan hij deze later gebruiken om toegang te krijgen tot dit account. Op deze manier kan hij de resultaten van de acties van het slachtoffer bekijken zoals persoonlijke informatie, verstuurde berichten, gebruikte zoektermen, etc.

Cross-Site Script Inclusion

Onderzoek door Lekies et al. toonde aan dat dynamisch gegenereerde JavaScript op verschillende manieren uitgebuit kan worden door een aanvaller [18]. Hoewel de broncode van een cross-origin script wordt afgeschermd, wordt het script wel uitgevoerd in zijn nieuwe context. Indien deze scripts gegenereerd zijn op basis van persoonlijke informatie vanwege een geauthenticeerde request, is het lucratief



FIGUUR 2.4: Voorbeeld van een Login CSRF-aanval op `shop.com`.

voor de aanvaller om bepaalde variabelen of parameters te achterhalen. Lekies et al. beschrijven verschillende manieren waarop de aanvaller te werk zou kunnen gaan. Een script dat uitgevoerd wordt in hetzelfde HTML-document heeft toegang tot alle globale variabelen van het cross-site script. Zo hoeft de aanvaller maar gewoon de waarden van deze variabelen in te lezen door middel van zijn zelf geschreven script. De aanvaller zou ook methoden uit de globale API van JavaScript kunnen herdefiniëren. Indien het cross-site script deze gebruikt, kan de parameter voor deze methode-oproep gelekt worden. Ten slotte kan er ook nog ingespeeld worden op de *prototype chain* [21], eigen aan JavaScript. Een object in JavaScript erft over van zijn prototype. Wanneer er een methode van een object wordt opgeroepen, en die niet is gedefinieerd voor dat object, dan wordt er gezocht naar die methode in het prototype van dat object. Dit proces herhaalt zich totdat de op te roepen methode is gevonden, of er geen prototype meer beschikbaar is (aangeduid door `null`). Als het prototype van een object globaal gedefinieerd is, dan kan de aanvaller functies van dit prototype overschrijven. Zo kan hij ervoor zorgen dat er parameters gelekt worden bij de oproep van de methode. Dit is heel bruikbaar bij *built-in* objecten in JavaScript zoals Array, String en Function. Omdat veel van hun methoden gedefinieerd zijn in hun prototypen, kan de aanvaller via het prototype het gedrag van opgeroepen functies veranderen.

Terada onderzocht nieuwe XSSI-aanvalstechnieken die gericht zijn op JSON- en CSV-bestanden, genaamd *identifier based XSSI attacks* [31]. Men slaagde er in om data te lekken door deze bestanden in te te laden door middel van de <script>-tag. Dit gebeurde bijvoorbeeld op basis van *error handling* van ongedefinieerde variabelen.

Timing attacks

Bortz en Boneh introduceerden de cross-site timing attack, een combinatie van cross-site aanval en de timing attack [4]. De impliciete authenticatie in de cross-site request zorgt ervoor dat gebruikersspecifieke informatie gelekt wordt door de tijd die nodig is om een dynamisch gegenereerde webpagina te versturen. Men is er onder andere in geslaagd om op deze manier te achterhalen hoeveel voorwerpen er zich bevonden in een online winkelkarretje van een gebruiker op een commerciële website. Volgens Bortz en Boneh kan deze techniek toegepast worden door gebruik van iframes, maar gebruik van de <img>-tag gaf hen de beste resultaten. Via de <img>-tag verwees men naar een bepaalde resource, waardoor deze werd opgehaald door de webbrowser. De hiervoor benodigde tijd werd berekend door een client-side side script. Als men dit proces een aantal keer herhaalde, kon men ook de ruis aanwezig in deze tijdsmetingen voldoende reduceren. Bovendien slaagde men er in het hele proces onzichtbaar te maken voor de gebruiker en hoeft men de gebruiker maar voor enkele seconden af te leiden om de metingen te voltooien.

Ook voor het geval dat er geen gebruik kan worden gemaakt van JavaScript, bespraken Bortz en Boneh een alternatief dat een timing attack toch uitvoerbaar kan maken. Volgens de paper laadt de webbrowser tags zoals de <link>- en <script>-tag namelijk volledig in alvorens over te gaan naar het volgende element. Als men hiermee verwijst naar de doelpagina, en men dit tussen twee andere elementen plaatst die

verwijzen naar de website van de aanvaller, dan kan men op deze manier ook een timing attack uitvoeren. Een voorbeeld hiervan wordt getoond in listing 2.3. Door via de eerste en laatste <script>-tag te verwijzen naar zijn eigen website, kan de aanvaller het tijdstip vóór de request naar `example.com` geïnitieerd wordt en het tijdstip nadat de daaropvolgende response behandeld is, bepalen. Het verschil tussen deze twee tijdstippen is dan afhankelijk van de benodigde tijd voor de referentie naar de doelpagina. Deze aanval probeerden we te simuleren in sectie 5.2.3, maar omdat deze resources tegenwoordig asynchroon worden ingeladen, slaagden we hier niet in.

Listing 2.3: Voorbeeld van een timing attack zonder gebruik van JavaScript.

```
<script src="https://attacker.com/time.php?timing=start" ></script>
<script src="https://example.com" ></script>
<script src="https://attacker.com/time.php?timing=stop" ></script>
```

De browser-based timing attack, geïntroduceerd door Van Goethem et al. [10], maakte komaf met de nadelige invloeden van de netwerksnelheid op de vorige aanval. De tijdsmeting start hier immers pas na het downloaden van de resource, zodat deze volledig onafhankelijk is van de netwerksnelheid. Men mat bijvoorbeeld hoe lang het duurde voor de webbrowser om een cross-site HTML-document te parsen als multimedia resource. Dit gebeurde met behulp van de <audio>- en <video>-tag samen met de tijdsmeting in JavaScript. Er wordt een `error`-event afgevuurd door de webbrowser indien de resource niet geparst kan worden als multimedia resource. De grootte van het HTML-document beïnvloedt echter de tijd die de webbrowser nodig heeft om dit event af te vuren. Hier maakte men gretig gebruik van door op basis van events de tijdsmeting te starten wanneer de resource gedownload is en te stoppen wanneer het `error`-event wordt afgevuurd. Dit zorgde ervoor dat informatie over de grootte van het HTML-document gelekt kon worden. Andere besproken technieken in deze paper zijn gebruik van script parsing, en caching door middel van ApplicationCache [22] en Service Workers [25]. Dat deze aanval toegepast kan worden op het web, toonde men aan met enkele experimenten. Zo werd er bijvoorbeeld aangetoond dat op basis van deze aanval achterhaald kon worden of een Facebook-gebruiker deel uitmaakt van een bepaalde demografische doelgroep.

Size-exposing attacks

Van Goethem et al. introduceerden nieuwe manieren om size-exposing attacks uit te voeren [11]. Zoals bij de andere aanvallen wordt er hier gebruik gemaakt van een geauthenticeerde cross-site request om een bepaalde resource te bemachtigen. Vervolgens probeert de aanvaller de grootte van deze resource te achterhalen om zo persoonlijke informatie over het slachtoffer te weten te komen. Van Goethem et al. vonden verschillende manieren om via APIs die beschikbaar worden gesteld door de webbrowsers op precieze wijze deze grootte vast te stellen en zo de SOP te omzeilen.

Een eerste methode maakt gebruik van *per-site* quota's. Aan de hand van de Fetch API en Cache API (beiden onderdeel van de Service Worker API [25]) kunnen cross-site resources worden opgehaald en opgeslagen. Er wordt echter gebruik

gemaakt van *per-site* quota's zodat één enkele website niet alle geheugenruimte kan innemen. Jammer genoeg kan dit gebruikt worden door een aanvaller om de exacte grootte van een resource te achterhalen, dit gebeurt als volgt. De website van de aanvaller zorgt ervoor dat de *per-site* quota bereikt is. Met andere woorden, er kan geen enkele resource meer in de cache opgeslagen worden door de website van de aanvaller. Nu kan de aanvaller, via zijn eigen website, één byte vrijmaken in deze cache en proberen de cross-site resource toe te voegen. Dit proces wordt herhaald tot het opslaan van die cross-site resource slaagt. Uiteindelijk zal de aanvaller dus weten hoeveel bytes er nodig zijn om de cross-site resource op te slaan.

Op een soortgelijke manier kan de globale quota gebruikt worden. Dit quota geeft aan hoeveel de webbrowser mag opslaan in de context van het besturingssysteem. Indien de quota is bereikt, zal de webbrowser de opslag van de minst recent gebruikte website verwijderen. Hier kan de aanvaller aan zien of de quota bereikt is. Door de *per-site* quota is het wel moeilijker om de globale quota te bereiken. De aanvaller zal immers verschillende domeinen moeten gebruiken om per domein onder de *per-site* quota te blijven, maar met alle domeinen samengeteld aan de globale quota te komen. Bij deze tactiek zal de aanvaller uiteindelijk ook de precieze grootte van een cross-origin resource kunnen achterhalen.

De aanvaller kan ook misbruik maken van de aangeboden Quota Management API [39] en Storage API [37]. Deze APIs zijn bestemd voor ontwikkelaars om meer informatie te verkrijgen over de opslag die gebruikt wordt door hun website. De aanvaller kan hier echter gemakkelijk mee checken hoe groot een cross-site resource is. Hij kan gewoon de huidige hoeveelheid van gebruikte opslag opvragen, hierna de cross-site resource opslaan en vervolgens opnieuw deze hoeveelheid opvragen. Het verschil tussen beide hoeveelheden is dan de grootte van de cross-site resource.

Van Goethem et al. toonden aan dat deze aanval kan toegepast worden op enkele populaire webapplicaties. Zo zou een Twitter-gebruiker via deze aanval geïdentificeerd kunnen worden. Daarnaast werd ook aangetoond dat het mogelijk is om de medische condities van een WebMD-gebruiker te achterhalen. Aan de hand van zoektermen die ingegeven kunnen worden bij web-based e-mailapplicatie zou ook persoonlijke informatie gelekt kunnen worden. Ten slotte slaagde men er ook in om na te gaan of een persoon lid is van een groepsgesprek op de populaire berichtendienst Telegram.

2.3.2 Verdediging gericht op specifieke cross-site aanvallen

In deze sectie bespreken we verdedigingsmaatregelen die voorgesteld zijn tegen specifieke cross-site aanvallen.

Login CSRF

Het onderzoek van Barth et al. ging na of bestaande CSRF-veiligheidsmaatregelen voldoende beschermen tegen Login CSRF [3]. Mits de implementatie rekening houdt met deze aanval, kunnen deze aanvallen inderdaad dienst doen ter bescherming tegen Login CSRF. CSRF-tokens zijn geschikt om ingezet te worden tegen Login CSRF, maar helaas wordt Login CSRF vaak vergeten bij de implementatie hiervan. Het is

hier immers de bedoeling dat de gebruiker al een session cookie toegewezen krijgt voordat hij zich ingelogd heeft, maar dit wordt vaak niet toegepast. Op deze manier kan er dan voor een CSRF-token validatie gezorgd worden op basis van het inloggen via bijvoorbeeld een HTML-form. Login CSRF zal hier dus niet meer toegepast kunnen worden omdat de aanvaller geen toegang heeft tot de CSRF-token.

Ook het checken van de `Referer`-header is volgens Barth et al. geschikt ter verdediging tegen Login CSRF. Men onderzocht in het bijzonder de mate waarin deze header gestript wordt in legitieme requests, om de bruikbaarheid te affirmeren. Uit dit onderzoek bleek dat de `Referer`-header vaker gestript wordt over HTTP dan over HTTPS. Strikte `Referer`-checking zou dan niet praktisch zijn voor requests over HTTP, omdat te veel gebruikers uitgesloten zullen worden. Aan de andere kant zou dit wel haalbaar zijn voor requests over HTTPS. Het ging hier namelijk om een relatief klein aantal instanties waarbij deze header gestript werd. Omdat login requests vooral verstuurd worden over HTTPS, lijkt `Referer`-checking volgens Barth et al. dus geschikt ter gebruik tegen Login CSRF.

Ook aangepaste HTTP-headers zijn geschikt volgens Barth et al., mits hier rekening mee gehouden wordt bij het inloggen. Zoals we al aanhaalden, kunnen aangepaste HTTP-headers enkel gedragen worden door same-site requests geïnitieerd door XHR (zonder expliciete toestemming via de `Access-Control-Allow-Headers`-header). Deze header werd in een experiment voor nagenoeg alle gebruikers meegestuurd, waardoor er zo goed als geen gebruikers buitengesloten werden. Men kan deze techniek gebruiken indien de logingegevens verstuurd worden door middel van XHR.

Cross-Site Script Inclusion

Ter bescherming tegen XSSI, stellen Lekies et al. voor om JavaScript-code strict te scheiden van gebruikersspecifieke gegevens [18]. Het is dus veiliger om deze gegevens te lezen met behulp van CORS, dan via JSONP. Het statische script kan zo nog altijd de gebruikersspecifieke informatie inlezen met XHR. In dit geval kunnen enkel de originele website of websites die expliciet op een white-list staan aan de gevoelige gegevens. Via de uitgevoerde JavaScript kan er ook geen informatie meer gelekt worden, omdat al deze informatie zich bevindt in niet uitvoerbare bestanden.

Naast de bovenstaande oplossing, bespreken Lekies et al. ook nog enkele situaties waarin een XSSI-aanval niet effectief is. Soms worden er onvoorspelbare URLs gebruikt om een dynamisch script op de halen. Zo een URL bevat dan bijvoorbeeld een unieke parameter, afhankelijk van de gebruiker, zoals de session id. Omdat de aanvaller geen toegang heeft tot deze unieke parameter, kan hij deze URL niet gebruiken voor een XSSI-aanval. Daarnaast kan strikte `Referer`-checking ook helpen tegen deze aanval.

Timing attacks

Als defensieve maatregel tegen cross-site timing attacks, ontwikkelden Bortz en Boneh een Apache-module die ervoor zorgt dat de webserver elke request beantwoordt in een

zo uniform mogelijke tijdspanne. [4]. Voor webservers die *chunked transfer encoding* gebruiken, worden ook de chunks op uniforme tijdstippen verstuurd. Een chunk kan enkel verzonden worden op een tijdstip dat een veelvoud is van een ingestelde waarde. Hoe groter deze waarde, hoe minder informatie er gelekt wordt. Bortz en Boneh halen ook nog aan dat dit probleem opgelost zou kunnen worden door modificaties aan de webbrowser, al zou het heel omslachtig zijn om alle manieren waarop tijd gemeten kan worden te blokkeren. Ook vermelden ze dat het toevoegen van een willekeurige, artificiële vertraging aan elke request de aanval niet kan stoppen, maar enkel kan vertragen. Schinzel vergelijkt in zijn onderzoek ook enkele mogelijkheden zoals de voorgenoemde constante responsetijd en willekeurige, artificiële vertraging, maar ook een constante en onvoorspelbare vertraging [28].

De verdedigingsmaatregelen tegen cross-site attacks kunnen meestal niet hergebruikt worden tegen browser-based timing attacks. In de paper van Van Goethem et al. focust men de mogelijke verdediging op het onderscheppen van geauthenticeerde cross-site requests die door de aanvaller geforceerd worden [10]. Meer bepaald stelt men hier een oplossing voor door het checken van de `Referer`-header. Indien de `Referer`-header van een request niet aanwezig is of een onvertrouwd domein aangeeft, dan zal een vervangende webpagina verstuurd worden als respons. Op deze webpagina wordt dan de originele content ingeladen via een geauthenticeerde XHR. Deze request zal de `Origin`- en `Referer`-header dragen zodat de webserver ziet dat het om een legitieme request gaat. Op deze manier kunnen geauthenticeerde cross-site requests niet meer gebruikt worden door de aanvaller om staatsafhankelijke informatie te bemachtigen. Dit moeten we wel nuanceren door het probleem omtrent gebruikers die de `Referer`-header strippen uit hun requests, zoals ook aangehaald wordt in de paper.

Size-exposing attacks

Omdat de grootte van cross-site resources gelekt wordt via de webbrowser APIs, stellen Van Goethem et al. voor om hier *virtual padding* op toe te passen [11]. Dit wordt gebruikt om de grootte van resources te verbergen door het toevoegen van een onvoorspelbaar aantal bytes. Dit is niet zomaar een willekeurig aantal bytes omdat de aanvaller dan nog altijd terug kan vallen op statistische aanvallen. Men verhoogt telkens het benodigde aantal bytes met de hash van de unieke id van de fetch-operatie uit de Fetch API [19]. Dit aantal wordt dan uiteindelijk afgerond op een bepaald aantal bytes. Het resultaat hiervan is de resource-grootte die de aanvaller zal kunnen zien, verschillend van de werkelijke grootte.

Daarnaast wordt er in diezelfde paper ook aangehaald dat het onderscheppen van illegitieme requests een verdienstelijke piste kan zijn ter verdediging tegen deze aanval. Onder andere verdedigingsmechanismen die we al bespraken in deze thesis kwamen hier aan bod.

2.3.3 Algemene verdediging tegen cross-site aanvallen

Met een algemene verdediging bedoelen we hier een verdediging die niet specifiek bedoeld is voor één van de besproken aanvallen, maar kan toegepast worden in het algemeen tegen cross-site aanvallen. We focussen hier op client-side verdediging. Er bestaan veel webbrowserextensies die een extra laag van bescherming bieden tegen cross-site aanvallen. Net zoals bij de same-site cookie, wordt er hier ook telkens client-side de beslissing genomen om bepaalde informatie wel of niet mee te sturen met een cross-site request. Zij het nu de hele cross-site request die wordt tegengehouden, of enkel de delen die zorgen voor impliciete authenticatie. We bespreken hier kort de werking van RequestRodeo en CsFire als voorbeeld van client-side bescherming tegen CSRF-aanvallen.

RequestRodeo

De allereerste client-side verdediging tegen cross-site aanvallen is RequestRodeo [16]. RequestRodeo is een lokale proxy op de computer van de gebruiker die elke HTTP-request en -response, tussen webbrowser en webserver, onderschept. Omdat RequestRodeo niet geïntegreerd is in een webbrowser, kan het geen gebruik maken van de API die aangeboden wordt door webbrowsers. Zo zal het bijvoorbeeld moeten achterhalen of een request same-origin is door HTTP-responses te manipuleren.

Voor elke HTTP-request bepaalt RequestRodeo of deze impliciete authenticatie mag dragen. Indien een request voortkomt uit de interactie met een webpagina (bv. het klikken op een link, het indienen van een HTML-form of via JavaScript) en het om een same-origin request gaat, dan wordt de request in zijn oorspronkelijke vorm verder gestuurd naar de webserver. Indien de request hier niet aan voldoet, wordt de impliciete authenticatie in de HTTP-headers als onvoldoende bevonden. Impliciete authenticatie in de vorm van een cookie wordt dan gewoon verwijderd uit de request en verzonden naar de webserver. Impliciete authenticatie in de vorm van een `Authentication`-header wordt ook genegeerd, maar via de proxy wordt de gebruiker wel nog gevraagd om expliciete authenticatie. Dit gebeurt door middel van een 401 HTTP-statuscode die ervoor zorgt dat de gebruiker een pop-upvenster wordt getoond met de vraag om zich te authenticeren. Na de expliciete authenticatie zal de proxy in het vervolg impliciete authenticatie toelaten via de `Authentication`-header voor requests afkomstig naar dezelfde URL vanaf die webpagina. De gebruiker wordt aldus op de hoogte gebracht van een authenticatie die mogelijks ongewenst zou kunnen zijn en heeft steeds de gelegenheid hiervan af te zien. Om de proxy zo weinig mogelijk de interactie tussen gebruiker en webapplicatie te laten verstoren, zorgden de ontwikkelaars ervoor dat verdachte requests niet helemaal geblokt werden. Zo wordt de functionaliteit van webapplicaties die geen authenticatie vereisen niet gehinderd.

CsFire

CsFire [5, 26] is een gratis browserextensie ontwikkeld door iMinds en KU Leuven die de gebruiker beschermt tegen cross-site aanvallen. In tegenstelling tot RequestRodeo

modificeert CsFire enkel HTTP-requests. Als browserextensie heeft CsFire namelijk de mogelijkheid om te achterhalen van welke origin een request afkomstig is, zonder HTTP-responses te manipuleren. Voor elke cross-site request die de webbrowser wilt maken, bepaalt CsFire of deze request schadelijk zou kunnen zijn voor de gebruiker. CsFire treedt hier dan gepast tegen op door de request toe te laten of te blokkeren, maar ook door in bepaalde gevallen cookies of andere authenticatie-informatie te verwijderen. Deze beslissing maakt CsFire op basis van verschillende policies, waarvan er één door de ontwikkelaars online ter beschikking wordt gesteld en één die door de gebruiker zelf in te stellen is. CsFire is beschikbaar voor Google Chrome en Mozilla Firefox en bereikt dus een groot publiek.

De functionaliteit van de same-site cookie kan vergeleken worden met die van browser-extensies zoals CsFire. Door middel van een extra cookie-attribuut weet de browser wanneer hij de cookie wel of niet mag meesturen met een cross-site request. Het extra attribuut wordt bepaald door de website die de cookie opstuurt. Dit kan gezien worden als voordeel ten opzichte van de browserextensies omdat de website dit zelf volledig in handen heeft terwijl er server-side geen extra implementatie nodig is. In het volgende hoofdstuk gaan we dieper in op de eigenlijke werking van de same-site cookie.

Hoofdstuk 3

Same-site cookies

In dit hoofdstuk bespreken we de same-site cookie. We gaan dieper in op de werking beschreven door de Internet Draft (sectie 3.1) en de moeilijkheden bij de adoptie ervan (sectie 3.2). Omdat de Internet Draft bepaalde vragen onbeantwoord liet, onderzoeken en toetsen we de werking van de same-site cookie nader (sectie 3.3). Ten slotte bespreken we de crawl die we uitvoerden om de huidige staat van het web qua cross-site requests te peilen (sectie 3.4). Dit doen we om in te spelen op gevonden gebruiken met een prototype om de adoptie van same-site cookies te vergemakkelijken.

3.1 De same-site cookie

In de oorspronkelijke Internet Draft uit 2014 stond de same-site cookie nog bekend onder de naam *first-party cookie* [35]. Inmiddels zijn er enkele aanpassingen gebeurd, waaronder de naamswijziging, met als resultaat de meest recente Internet Draft over de same-site cookie [36]. Ondertussen wordt de same-site cookie al ondersteund door de open-source webbrowser Chromium. Als logisch gevolg bieden de afgeleiden van Chromium, de webbrowsers Google Chrome (vanaf versie 51, uitgebracht op 31 mei 2016) en Opera (vanaf versie 39, uitgebracht op 2 augustus 2016) ook ondersteuning [30]. Voor andere grote webbrowsers zoals Mozilla Firefox, Microsoft Edge en Safari is dit nog niet het geval. De same-site cookie is echter achterwaarts compatibel, zo worden same-site cookies als gewone cookies behandeld door deze laatste groep van webbrowsers.

De same-site cookie kan worden geïnterpreteerd als een gewone cookie met een extra attribuut. Dit extra attribuut is het `SameSite`-attribuut. Er kunnen twee verschillende waarden toegewezen worden aan dit attribuut: *strict* en *lax*. Een cookie waarvan het `SameSite`-attribuut de waarde strict heeft, noemen we een `SameSite-strict` cookie. Een cookie waarvan het `SameSite`-attribuut de waarde lax heeft, noemen we een `SameSite-lax` cookie. De same-site cookie kan op dezelfde manier worden ingesteld als een gewone cookie. Het extra attribuut wordt gewoon achteraan toegevoegd zoals te zien in listing 3.1 voor de Set-Cookie header en in listing 3.2 voor JavaScript.

Listing 3.1: Voorbeeld van een `SameSite-lax` cookie die wordt ingesteld via de `Set-Cookie`-header.

```
Set−Cookie: lax_cookie=123; Domain:example.com; Path=/; SameSite=lax
```

Listing 3.2: Voorbeeld van een `SameSite-strict` cookie die wordt ingesteld door middel van JavaScript.

```
document.cookie="strict_cookie=456; Domain:example.com; SameSite=strict"
```

Alle same-site cookies worden hoe dan ook meegestuurd met alle same-site requests. De same-site cookie Internet Draft beschrijft een same-site request als een verzoek waarin het domein van de doel-URL een exacte overeenkomst is met het domein in wiens context het verzoek verstuurd werd. Doch, het domein wordt op een specifieke manier gedefinieerd in de Internet Draft. In sectie 3.3.3 gaan we hier dieper op in.

De waarde van het `SameSite`-attribuut schrijft voor hoe de webbrowser de same-site cookie moet behandelen. De `SameSite-strict` cookies worden nooit met cross-site requests meegestuurd, enkel met same-site requests. De `SameSite-lax` cookies worden onder bepaalde voorwaarden met cross-site requests meegestuurd. De vuistregel hier is dat de cross-site request een HTTP GET-request moet zijn die een top-level navigatie veroorzaakt. In sectie 3.3.1 bekijken we dit meer gedetailleerd en onderzoeken we welke same-site cookies nu eigenlijk met welke cross-site requests worden meegestuurd.

Het doel van de same-site cookie is het bieden van een extra laag van bescherming tegen CSRF-aanvallen, XSSI-aanvallen en cross-site timing attacks. Wanneer er geen authenticerende cookie wordt meegestuurd met de cross-site request, worden deze aanvallen immers onschadelijk gemaakt.

3.2 Struikelblokken bij de adoptie en kwetsbaarheden

De adoptie van de same-site cookie door websites gebeurt moeizaam, daar er nog maar twee populaire webbrowsers ondersteuning bieden sinds bijna een jaar. Tijdens dit onderzoek hebben we nog maar één website ontdekt die gebruik maakt van same-site cookies, namelijk Dropbox. In een eerdere blog bespraken ze al verschillende mogelijkheden waarop de same-site cookie geadopteerd zou kunnen worden [6]. Bij de afweging wordt onder andere het breken van functionaliteit door de `SameSite-strict` cookie en de omzeilbaarheid van de `SameSite-lax` cookie aangehaald. Voor grotere websites zoals deze is het niet zo vanzelfsprekend om gebruik te beginnen maken van same-site cookies. Heel wat functionaliteit van hedendaagse websites berust namelijk op cross-site requests. Administrators zullen moeten overwegen welke cookies same-site worden, en in het bijzonder welke same-site cookies `SameSite-strict` of `SameSite-lax` worden. Het onbezonnen toepassen van same-site cookies zou bepaalde functies van de website kunnen breken. In hoofdstuk 4 beschrijven we een prototype dat de adoptie van same-site cookies tracht te vergemakkelijken voor administrators.

We evalueren dit prototype op basis van performantie in sectie 5.1 en testen het aan de hand van een aantal moderne webaanvallen in sectie 5.2.

3.3 Verkennend onderzoek naar de werking van de same-site cookie

Omdat de Internet Draft over bepaalde zaken geen uitsluitsel gaf of omdat we bepaalde zaken wilden toetsen, hebben we enkele experimenten verricht in een verkennend onderzoek naar het gedrag van de same-site cookie.

3.3.1 Analyse van cookie dragende cross-site requests

Er bestaan heel wat verschillende manieren waarop informatie gelekt kan worden op een HTML-pagina. De GitHub repository HTTPLeaks[1] tracht al deze verschillende manieren te verzamelen in één HTML-pagina. Deze HTML-pagina hebben we gebruikt om te onderzoeken welke same-site cookies nu precies met welke cross-site requests worden verzonden door de webbrowser.

Voor dit experiment maakten we gebruik van een Apache-server. We pasten de HTTPLeaks-pagina zo aan dat er gelekt werd naar deze Apache-server. Verder stelden we de logging van de Apache-server in zodat bij elke request ook werd vastgelegd welke cookies werden meegestuurd. Dan hebben we drie cookies aangemaakt: een generische cookie (zonder `SameSite`-attribuut), een `SameSite-Lax` cookie en een `SameSite-strict` cookie. Bij een bezoek aan de HTTPLeaks-pagina verzond de browser een hele reeks cross-site requests die dan ontvangen werden door onze Apache-server. Aan de hand van een script werden de serverlogs geïnterpreteerd en in een CSV-bestand geschreven.

De belangrijkste resultaten worden weergegeven in tabel 3.1. De volledige verzameling van resultaten is terug te vinden in tabel A.1 van bijlage A. Per request staat er aangegeven om welk lek het ging en welke cookies werden meegestuurd. Dit experiment hebben we uitgevoerd voor de webbrowsers Google Chrome (versie 56.0.2924.87), Opera (versie 43.0.2442.806) en Mozilla Firefox (versie 51.0.1). De resultaten voor Google Chrome en Opera waren identiek en zijn samengevoegd in één kolom. Een vinkje geeft aan dat de cookie werd meegestuurd met de cross-site request en een kruisje geeft aan dat de cookie niet werd meegestuurd. Voor de vakjes met een horizontaal streepje werd er geen request geregistreerd door de Apache-server. De lekken waarvoor er bij geen enkele webbrowser een request werd geregistreerd, zijn niet opgenomen in de tabel. De reden voor ontbrekende generische cookies of ontbrekende requests vonden we terug in de vorm van bugs, gebrek aan ondersteuning of inconsistentie in functionaliteit tussen de verschillende webbrowsers.

Firefox ondersteunde same-site cookies niet bij de uitvoering van dit experiment. Er wordt geen onderscheid gemaakt tussen generische cookies en same-site cookies door Firefox zoals uit de tabel afgeleid kan worden. De lekken met een asterisk

[1] https://github.com/cure53/HTTPLeaks

vereisten interactie van de gebruiker, door de klikken op een HTML-element. Dit hebben we gedaan voor elk zichtbaar element met de tekst “CLICKME”.

Tabel 3.1: Cookies meegestuurd met cross-site requests.

Webbrowser-engine	Blink			Gecko		
Webbrowser(s)	Google Chrome Opera			Mozilla Firefox		
Cookie	gen	lax	strict	gen	lax	strict
a-href*	✓	✓	✗	✓	✓	✓
iframe-src	✓	✗	✗	✓	✓	✓
img-src	✓	✗	✗	✓	✓	✓
link-prerender	✓	✓	✓	-	-	-
meta-refresh	✓	✓	✗	✓	✓	✓

Conform met de specificatie van de same-site cookies wordt voor de <img>- en <iframe>-tags geen same-site cookie meegestuurd. De `SameSite-lax` cookie wordt dan wel weer meegestuurd bij de klik op een hyperlink (a-href) en bij een meta-refresh. Deze elementen zorgen immers voor een top-level navigatie. De resultaten voor het prerender-element zijn echter niet zoals verwacht, maar het bleek hier uiteindelijk te gaan om een bug.

Prerender-bug

Onze analyse toonde aan dat bij gebruik van de <link rel="prerender"href="...">-tag de `SameSite-Lax` cookies en de `SameSite-strict` cookies meegestuurd worden met een cross-site request. Dit is terug te vinden in tabel 3.1 in de rij van `link-prerender`. Het prerender-element zorgt ervoor dat een specifieke pagina achter de schermen kan worden ingeladen, zodat wanneer de pagina wordt opgevraagd door de gebruiker, er zo weinig mogelijk vertraging plaatsvindt [12]. Dit proces is volledig onzichtbaar voor de gebruiker. Hoewel het hier niet gaat om een top-level navigation, wordt er in de Internet Draft wel een uitzondering gemaakt voor `SameSite-Lax` cookies. Volgens de Internet Draft mogen `SameSite-lax` cookies dus meegestuurd worden met een cross-site request die geïnitieerd werd door middel van het prerender-element. Dit staat zo echter niet gespecifieerd voor `SameSite-strict` cookies, hoewel deze hier wel meegestuurd worden met cross-site requests. Hiervoor hebben we een bug report ingediend bij het Chromium Project, waar de bug erkend werd. Op het moment van schrijven was de bug nog niet hersteld.

3.3.2 Server-side redirect versus client-side redirect

Redirects kunnen server-side of client-side zijn. Server-side redirects verwijzen de webbrowser door naar een ander webadres op basis van HTTP-headers. In de response van de server is de Location-header dan ingesteld op de nieuwe doel-URL en wordt er ook een HTTP redirect code (3xx) meegegeven [7]. Bij ontvangst van deze response zal de webbrowser automatisch de URL opvragen die gespecifieerd is

in de Location-header. Client-side redirects verwijzen de webbrowser door via een client-side script. Dit kan onder meer door gebruik van de <meta>-tag in HTML (zie listing 3.3) of het `window.location`-attribuut in JavaScript (zie listing 3.4). Een iframe kan een client-side redirect simuleren door de hele pagina in beslag te nemen.

LISTING 3.3: Een client-side redirect door middel van de <meta>-tag in HTML.

```
<html>
    <head>
        <meta http-equiv="refresh"
            content="0; url=https://example.com/goal" />
    </head>
    <body></body>
</html>
```

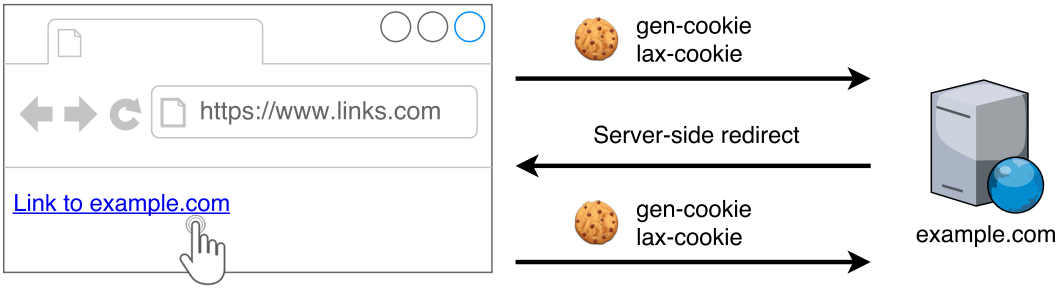
LISTING 3.4: Een client-side redirect door middel van het `window.location`-attribuut in JavaScript.

```
<html>
    <head></head>
    <body>
        <script> window.location.replace('http://example.com/goal')
        </script>
    </body>
</html>
```

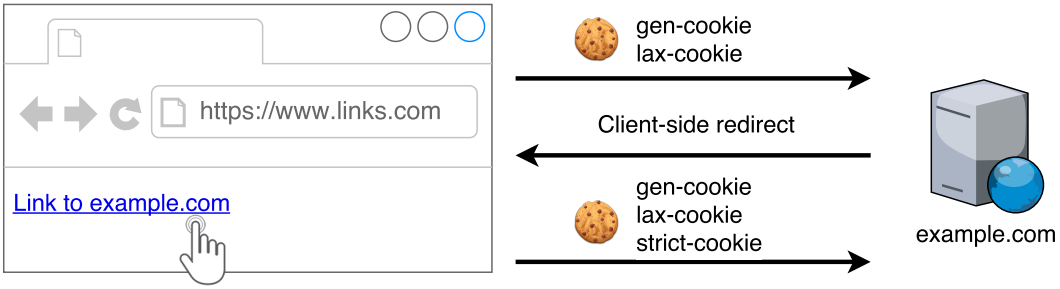
We ondervonden dat het gedrag van same-site cookies significant verschilt tussen server-side en client-side redirects. In figuur 3.1 wordt de gang van zaken bij een server-side redirect afgebeeld. Stel dat de gebruiker drie cookies heeft voor het domein `example.com`. Dit zijn gen-cookie (een generische cookie), lax-cookie (een `SameSite-Lax` cookie) en strict-cookie (een `SameSite-strict` cookie). De gebruiker surft naar de website `links.com` en klikt daar op een hyperlink naar een pagina op `example.com`. Deze request bevat de cookies gen-cookie en lax-cookie, conform met de specificaties van de same-site cookie. De webbrowser beantwoordt deze request met een server-side redirect waardoor de webbrowser automatisch de nieuwe doel-URL opvraagt die gespecifieerd is in de Location-header. Bij deze laatste request worden dezelfde cookies meegestuurd.

Hetzelfde scenario waarbij een client-side redirect wordt gebruikt in plaats van een server-side redirect staat afgebeeld in figuur 3.2. Bij de uiteindelijke request naar de nieuwe doel-URL wordt hier de `SameSite-strict` cookie ook meegestuurd. Dit gebeurt omdat de webbrowser deze client-side redirect interpreteert als een same-site request waardoor hij als gevolg ook alle same-site cookies meestuurt. Dit zou uitgebuit kunnen worden door een aanvaller indien er een client-side direct is naar een pagina die toestandswijzigende requests aanneemt op basis van cookies.

Anderzijds zou de aanvaller ook zelf de client-side redirect kunnen injecteren in een webpagina van `example.com`.



Figuur 3.1: Voorbeeldscenario waarin enkel de generische cookie en `SameSite-lax` cookie worden meegestuurd na een server-side redirect ten gevolge van een cross-site request.



Figuur 3.2: Voorbeeldscenario waarin de `SameSite-strict` cookie wordt meegestuurd na een client-side redirect ten gevolge van een cross-site request.

Deze uitbuiting van client-side requests kan enkel gebeuren wanneer de webbrowser de pagina met de redirect interpreteert. Dit is bijvoorbeeld niet het geval wanneer de pagina opgehaald wordt voor de <img>-tag. Ook de <iframe>-tag op de webpagina van de aanvaller kan niet gebruikt worden omdat de client-side redirect gebeurt in de context van de website van de aanvaller. Hierdoor worden er geen same-site cookies meegestuurd. De aanvaller kan nog altijd gebruik maken van clickjacking door middel van een hyperlink of zelf een redirect uitvoeren op zijn eigen pagina. Dit maakt de aanval wel heel opvallend.

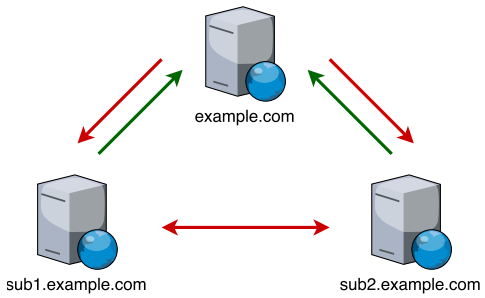
3.3.3 Cross-site requests tussen subdomeinen

In het kader van het verkennend onderzoek hebben we ook onderzocht hoe de webbrowser omgaat met same-site cookies in de context van subdomeinen. Dit gebeurt op een andere manier dan gestipuleerd wordt volgens RFC 6265 [1] omtrent het `Domain`-attribuut van cookies.

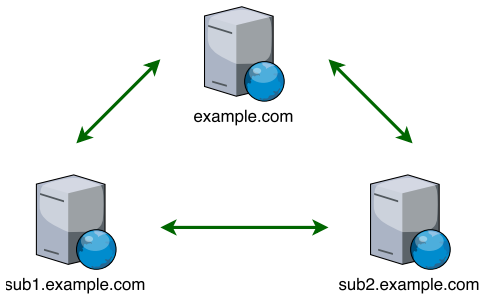
De regulering van het `Domain`-attribuut in de context van een voorbeeldscenario staat afgebeeld in figuur 3.3. De cookies zijn ingesteld door middel van de `Set-Cookie`-

headers in listing 3.5. Een groene pijl betekent dat er toegang verschaft wordt voor het ene domein tot cookies van het andere domein. Bij een rode pijl wordt er geen toegang verschaft. Toegang wilt zeggen dat het ene domein cookies van het andere domein kan creëren (op basis van het `Domain`-attribuut) en kan lezen. Zo heeft het domein `example.com` geen toegang tot cookies van zijn subdomeinen (sub1-cookie en sub2-cookie). De subdomeinen hebben daarentegen wel toegang tot cookies van `example.com` (example-cookie), maar geen toegang tot elkaars cookies.[2]

Domeinen worden anders vergeleken met elkaar volgens de same-site cookie Internet Draft. De Internet Draft vergelijkt cookies op basis van "*registrable domain*" [36]. Dit wordt gedefinieerd als de publieke suffix (bv. .com, .co.uk, .be) plus het label aan de linkerkant. Het "*registrable domain*" van `https://www.sub1.example.com` is bijvoorbeeld `example.com`. Op basis van dit criterium zullen dus alle cross-site requests in ons voorbeeldscenario same-site cookies dragen zoals afgebeeld in figuur 3.4. Een cross-site request van `sub1.example.com` naar `example.com` zal dus de cookie example-cookie meedragen. In de omgekeerde richting zal de cookie sub1-cookie meegestuurd worden. Cross-site requests tussen subdomeinen zullen ook de same-site cookies meedragen.



FIGUUR 3.3: Schematische weergave van de toegang tot cookies in de context van subdomeinen.



FIGUUR 3.4: Schematische weergave van de werking van same-site cookies in cross-site requests in de context van subdomeinen.

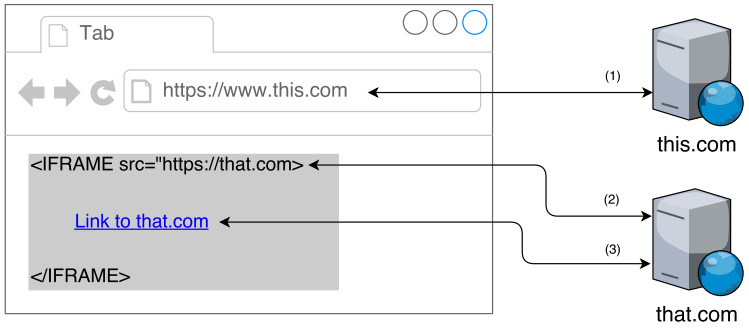
LISTING 3.5: De ingestelde cookies voor het voorbeeld in figuur 3.3 en figuur 3.4.

```
Set-Cookie: example-cookie=1; Domain=example.com; SameSite=strict
Set-Cookie: sub1-cookie=1; Domain=sub1.example.com; SameSite=strict
Set-Cookie: sub2-cookie=1; Domain=sub2.example.com; SameSite=strict
```

3.3.4 Andere inconsistenties

Zoals tabel 3.1 al toonde, worden er geen `SameSite-strict` cookies meegestuurd bij een cross-site request geïnitieerd door een hyperlink. Althans, dit is het geval wanneer

[2] Wanneer het `Domain`-attribuut van de cookie van `example.com` niet gespecifieerd zou zijn in de `Set-Cookie`-header, dan zouden de subdomeinen deze cookie niet kunnen lezen maar wel overschrijven.



Figuur 3.5: Voorbeeld waarbij de hyperlink in een iframe toch same-site cookies met een cross-site request meestuurt.

de gebruiker hiervoor een linkermuisklik gebruikt. Er zijn nog andere mogelijkheden om de gelinkte pagina te openen in de webbrowser. Zo kan de gebruiker ervoor kiezen om de pagina te openen in een nieuw tabblad of nieuw venster door het menu te openen met een rechtermuisklik. Dit kan ook aan de hand van een sneltoetscombinatie. We ontdekten dat bij het openen van een hyperlink in een nieuw tabblad of nieuw venster, de `SameSite-strict` wel wordt meegestuurd met de cross-site request. Deze cookie wordt dan weer niet meegestuurd als de hyperlink automatisch een nieuw tabblad of venster opent door middel van het `target`-attribuut.

Dit gedrag is gelijkaardig wanneer de link in kwestie zich bevindt op een webpagina van hetzelfde domein, maar omsloten is in een iframe op een ander domein. Dit scenario wordt afgebeeld in figuur 3.5. Een webpagina van `this.com` bevat hier een iframe die verwijst naar een webpagina van `that.com`. Die laatste webpagina bevat een hyperlink naar een andere webpagina van `that.com`. Net zoals hiervoor zal bij het openen van deze hyperlink door middel van een linkermuisklik geen `SameSite-strict` cookie meegestuurd worden met de cross-site request naar `that.com`. De webpagina waarnaar de link verwijst wordt dan geopend in de iframe weliswaar. Ook hier zal de `SameSite-strict` net wél weer meegestuurd worden bij het handmatig openen van deze link in een nieuw tabblad of venster. Hier is het echter ook mogelijk om hetzelfde te bereiken door de link automatisch te laten openen in een nieuw tabblad of venster bij een linkermuisklik door middel van het `target`-attribuut. In dit geval worden de `SameSite-strict` cookies van `that.com` ook meegestuurd wanneer de gebruiker op deze link klikt.

Men zou kunnen zeggen dat deze inconsistenties uitgebuit kunnen worden door een aanvaller (bv. clickjacking), maar daarnaast kunnen ze ook zorgen voor enige vorm van frustratie bij de gebruiker. De gebruiker merkt bijvoorbeeld dat hij in het ene scenario ingelogd blijft en in het andere niet. Daarom is het belangrijk om naar de websitebeheerders en gebruikers toe een zo consistent mogelijke functionaliteit aan te bieden.

3.4 Verkennend onderzoek naar de huidige staat van het web

We voerden een exploratief experiment uit om de huidige staat van het web wat betreft cross-site requests in kaart te brengen. Zo kunnen we nagaan op welke manier we de adoptie van same-site cookies zouden kunnen vergemakkelijken. We zijn vooral geïnteresseerd in welke websites veel cross-site requests ontvangen omdat dit een indicatie is dat de functionaliteit van de website sterk steunt op cross-site requests. Zo kunnen we dan toetsen of deze requests geauthenticeerd zijn en gebruikt worden om bepaalde acties te laten verrichten door de gebruiker. We vragen ons ook af of acties via same-site requests dan op dezelfde manier geauthenticeerd worden. Op deze manier gaan we dan na of deze websites baat hebben bij gebruik van same-site cookies.

We zijn gestart met het onderzoeken naar welke websites de meeste cross-site requests gemaakt worden. Met behulp van een crawler onderzochten we naar welke domeinen het meest gerefereerd wordt met cross-site requests. Hiervoor gebruikten we de Alexa Top 10.000. De crawler bezocht alle websites in deze top 10.000 van meest bezochte websites en sloeg alle referenties op die op deze websites gemaakt werden. Uit deze informatie stelden we dan een lijst op van alle domeinen gesorteerd op het aantal cross-site requests dat het domein ontving. Tabel 3.2 toont de eerste tien domeinen uit deze lijst, samen met het aantal unieke domeinen vanwaar er gerefereerd wordt en de categorie waartoe het gerefereerde domein behoort volgens Trendmicro[3].

De top 300 van onze lijst wordt gedomineerd door Google services, sociale media en vooral door advertentiemaatschappijen. Voor deze top 300 voerden we een manuele analyse uit om na te gaan of deze websites enkel cross-site gebruikt worden of ook andere functionaliteit aanbieden. Dit deden we door voor elke website die dit toeliet een account te registreren. Uiteindelijk waren het maar 18 websites waarop een

[3] https://global.sitesafety.trendmicro.com

Tabel 3.2: Top 10 van de meest gerefereerde websites uit ons onderzoek.

Rang	Gerefereerd domein	Aantal unieke refererende domeinen	Categorie
1	google-analytics.com	6768	Internet infrastructuur
2	doubleclick.net	6320	Advertising
3	google.com	5793	Zoekmachine
4	googleapis.com	5267	Computers / Internet
5	facebook.com	4663	Sociaal netwerk
6	gstatic.com	4201	Computers / Internet
7	facebook.net	4144	Sociaal netwerk
8	googlesyndication.com	3388	Zoekmachine
9	adnxs.com	3105	Computers / Internet
10	google.be	2859	Zoekmachine

account geregistreerd kon worden. Omdat deze websites ook telkens de mogelijkheid aanbieden om vanop deze website zelf accountinstellingen te wijzigen, wordt er buiten de cross-site functionaliteit ook same-site functionaliteit aangeboden. Deze functionaliteiten verliepen doorgaans wel via andere endpoints van de website, maar werden geauthenticeerd door dezelfde cookies.

We kunnen dit illustreren aan de hand van een voorbeeld uit deze 18 websites, namelijk Facebook. De functionaliteit van Facebook steunt veel op gebruik van cross-site requests zoals blijkt uit de resultaten van de crawl. Door een account aan te maken en bepaalde acties te verrichten, gingen we na welke geauthenticeerde acties op basis van same-site en cross-site requests gebeuren. Facebook laat bijvoorbeeld externe websites toe een 'vind-ik-leuk'-knop te gebruiken. Als een Facebook-gebruiker op deze knop klikt, registreert Facebook dat de gebruiker de gerefereerde Facebook-pagina of -post leuk vindt net zoals dit het geval zou zijn wanneer de gebruiker dit doet op de website van Facebook zelf. Dit gebeurt door middel van een cross-site request die geauthenticeerd wordt via cookies. De endpoint die hiervoor gebruikt wordt is `/plugins/like/connect`. We ontdekten dat een same-site request die gebruikt wordt om het wachtwoord te wijzigen op dezelfde manier geauthenticeerd wordt. Dit gebeurt echter via de endpoint `/ajax/settings/security/password.php`. Zo kunnen we algemener een onderscheid maken tussen functies die gebruik maken van geauthenticeerde same-site requests (bv. ten behoeve van wijzigen van persoonlijke informatie) en geauthenticeerde cross-site requests (bv. ten behoeve van het delen of plaatsen van reacties via externe websites). Omdat deze functies op dezelfde manier geauthenticeerd worden - door middel van dezelfde cookies - zijn ze beiden even kwetsbaar voor cross-site aanvallen. Dit terwijl de functies die same-site aangeboden worden typisch gevoeliger zijn dan functies die cross-site aangeboden worden.

Omdat de impactgrootte van een aanval op deze acties sterk kan verschillen, is het handig om deze acties ook te beveiligen op basis van deze verschillende impactgroottes. Same-site cookies kunnen gebruikt worden om impactvolle acties die gebeuren via geauthenticeerde same-site requests onmogelijk te maken via cross-site requests, en dus te beschermen tegen cross-site aanvallen. Omdat deze verschillende acties vaak door dezelfde cookies geauthenticeerd worden, is het vaak lastig om enerzijds niet de functionaliteit van de website aan te tasten en anderzijds de cookiestrategie zo weinig mogelijk te veranderen. Om dit op te lossen, ontwierpen we een server-side prototype dat we beschrijven in het volgende hoofdstuk. Dit prototype maakt handig gebruik van het feit dat verschillende endpoints worden gebruikt voor het aanbieden van bepaalde functionaliteiten.

Hoofdstuk 4

Ontwikkeling van het prototype

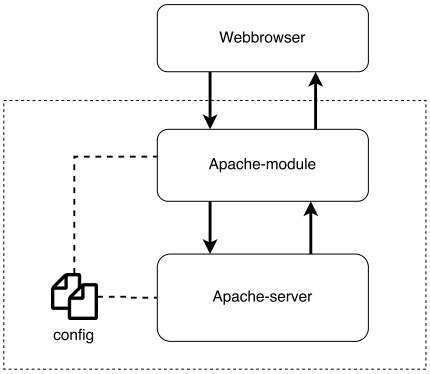
We hebben een prototype ontwikkeld dat tracht een groot aantal struikelblokken omtrent de adoptie van same-site cookies te verhelpen. In dit hoofdstuk beschrijven we het ontwerp en de werking van dit prototype (sectie 4.1). Vervolgens halen we ook enkele alternatieven aan die het probleem op een andere manier zouden kunnen oplossen (sectie 4.2).

4.1 Beschrijving van het prototype

Het prototype is ontwikkeld in de vorm van een server-side module. We kozen ervoor om dit server-side aan te pakken omdat we zo meer controle hebben over de voorwaarden waarop cookies worden geaccepteerd door de server. Zoals we hieronder uitleggen, bepalen we namelijk via de module in welke situaties bepaalde cookies geaccepteerd mogen worden. Deze server-side oplossing laat tevens toe dat er toch ondersteuning geboden kan worden aan applicaties waarvan de broncode niet gewijzigd kan worden. Voorbeelden hiervan zijn legacy-applicaties en contentmanagementsystemen zoals Wordpress en Joomla. In sectie 4.2.2 bespreken we ook een alternatieve client-side oplossing.

Met dit prototype willen we de webserveradministrator de mogelijkheid geven om same-site cookies op een uniforme en gemakkelijke manier toe te kunnen passen, zonder dat de implementatie van de website zelf gewijzigd hoeft te worden. Men zal met behulp van dit prototype dus in staat zijn om een website, zonder modificaties, te hosten op een webserver die gebruik maakt van de same-site cookie functionaliteit om deze website een extra laag van bescherming te bieden tegen cross-site aanvallen. Enkel de webserver zal geconfigureerd moeten worden, maar dit zal ook op een eenvoudige manier kunnen gebeuren. Het prototype doelt vooral op het beveiligen van session cookies omdat deze gebruikt worden ter authenticatie van een gebruikerssessie. Via de session cookie geeft het prototype een fijnere controle aan de administrator over welke acties geïnitieerd mogen worden door cross-site requests die geauthenticeerd zijn door deze session cookie. Door de website-implementatie af te schermen van same-cookie functionaliteit, hopen we de drempel voor bestaande websites te verlagen om same-site cookies te gaan gebruiken.

We kozen ervoor om dit prototype te ontwikkelen als Apache-module. Dit omdat Apache-webservers met 49,5% het sterkst vertegenwoordigd zijn op het web [32]. Met deze module kan een Apache-webserver gemakkelijk uitgebreid worden met het prototype. Daarnaast is het prototype ook gemakkelijk configureerbaar via de module-specifieke directieven in de configuratiebestanden van de Apache-server. Dit staat schematisch afgebeeld in figuur 4.1. Het prototype zou ook geïmplementeerd kunnen worden als module voor andere webservers zoals nginx, waardoor het concept dus breed toepasbaar is.



Figuur 4.1: Schematische voorstelling van de interactie tussen webbrowser, webserver en ontwikkelde Apache-module.

4.1.1 Het concept

We willen via deze module de administrator een meer verfijnde controle geven over session cookies met behulp van same-site cookies. Het doel is om een extra laag van bescherming te bieden tegen cross-site aanvallen, terwijl we de gebruiksvriendelijkheid en functionaliteit van de website zoveel mogelijk proberen te bewaren. Voor dat eerste zouden we elke session cookie `SameSite-strict` kunnen maken, voor dat laatste zouden we same-site cookies gewoonweg kunnen verwerpen. Het is de bedoeling om hier een goede afweging te maken tussen de twee.

Het concept van onze module begint bij het verdelen van de website op basis van de impact bij een aanval. De impact op een pagina die toelaat accountgebonden informatie te wijzigen of te verwijderen is bijvoorbeeld hoog. De impact op een pagina die enkel accountgebonden informatie weergeeft kan dan weer laag zijn. Voor het eerste geval zou een `SameSite-strict` of `Samesite-lax` session cookie een diepgaandere bescherming kunnen bieden. Deze cookie wordt immers niet of slechts in bepaalde gevallen meegestuurd bij een cross-site request, wat ongunstig is voor de slaagkans van een cross-site aanval. Voor het tweede geval zou dit echter onnodig kunnen zorgen voor minder gebruiksvriendelijkheid. Hier zou een generische cookie kunnen volstaan omdat de SOP ervoor zorgt dat de weergegeven informatie wordt afgeschermd van de aanvaller.

Net zoals we de website kunnen verdelen in bepaalde impactzones, kunnen we de session cookie opsplitsen in verschillende cookies met een specifieke autoriteit. Zo is een opgesplitste session cookie enkel geldig voor de impactzone waarvoor deze bestemd is. Een opgesplitste session cookie kan bijvoorbeeld een request authenticeren om een bericht te plaatsen, maar diezelfde cookie is niet voldoende om een request te authenticeren die persoonsgegevens wijzigt. De opsplitsing van deze session cookies is gebaseerd op het `SameSite`-attribuut van de same-site cookie. Op deze manier kunnen we de session cookie opsplitsen in drie verschillende session cookies. De eerste session cookie is er een zonder `SameSite`-attribuut en is bedoeld om requests te authenticeren met geen of weinig impact. De tweede session cookie, een `SameSite-lax` cookie, zou kunnen instaan voor de authenticatie van requests met een middelmatige impact. De laatste session cookie is een `SameSite-strict` cookie en doelt op authenticatie van requests met hoge impact. Deze verdeling zorgt ervoor dat cross-site aanvallen moeilijker uit te voeren zijn voor acties die een hoge impact hebben. Daarnaast zorgen we er ook voor dat de functionaliteit van acties met geringe impact niet lijden onder de hoge veiligheidsvoorwaarden die gelden voor acties met hoge impact. Het is bijvoorbeeld perfect mogelijk om met dezelfde session cookie een gebruiker toe te laten om accountsgegevens te bekijken en te wijzigen, maar deze laatste actie enkel toe te laten door middel van same-site requests.

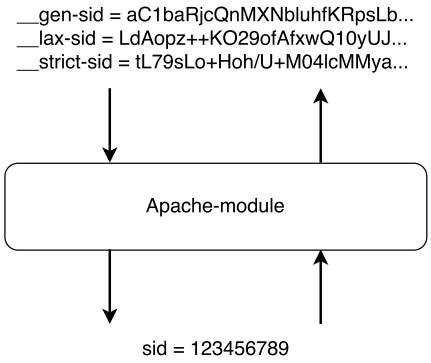
Opsplitsing door middel van encryptie

We hebben ervoor gekozen om het opsplitsen van de session cookie te realiseren door middel van encryptie. Hierdoor hoeft er geen extra sessiespecifieke informatie server-side opgeslagen te worden en kunnen de nieuwe cookiewaarden niet voorspeld worden door een aanvaller. De drie nieuwe session cookies komen voort uit de encryptie van de cookiewaarde van de originele session cookie. Ze zijn elk met een verschillende sleutel geëncrypteerd, namelijk de sleutel `key_gen`, `key_lax` of `key_strict`. Elk van de drie cookies krijgt een specifieke prefix en draagt een geëncrypteerde waarde van de oorspronkelijke cookie. Voor de prefixes haalden we onze inspiratie bij *cookie prefixes* [34]. Elke prefix impliceert de rol van de cookie:

- `__gen-` wordt gegeven aan de cookie zonder `SameSite`-attribuut, dit is de generische cookie die wordt geëncrypteerd en gedecrypteerd met de sleutel `key_gen`;
- `__lax-` is bestemd voor de cookie uitgebreid met het `SameSite`-attribuut lax, deze cookie is dus `SameSite-lax` en wordt geëncrypteerd en gedecrypteerd met de sleutel `key_lax`;
- `__strict-` wordt gekoppeld aan de cookie uitgebreid met het `SameSite`-attribuut strict, deze cookie is dus `SameSite-strict` en wordt geëncrypteerd en gedecrypteerd met de sleutel `key_strict`.

Figuur 4.2 stelt deze structuur voor door middel van een voorbeeld. De webbrowser ziet de drie geconverteerde cookies, terwijl de webserver enkel één session cookie

ziet, namelijk de session cookie die het zelf heeft ingesteld. De administrator stelt zelf in welke session cookie er telkens geconverteerd wordt door de module (in het voorbeeld is dit `sid`), alsook welke geconverteerde cookie geaccepteerd mag worden op welke plaats op de website. Wanneer de webserver deze cookie instelt via de `Set-Cookie`-header, converteert de module deze header. In plaats daarvan worden de drie verschillende cookies ingesteld. In de omgekeerde richting zal de module één cookie van dit drietal in de `Cookie`-header converteren, afhankelijk van de configuratie die de administrator heeft ingesteld. De andere twee cookies worden verwijderd uit de header. Indien de te converteren cookie zich niet in de `Cookie`-header bevindt, ontvangt de webserver de session cookie in kwestie niet. Dit kan bijvoorbeeld het geval zijn wanneer de cookie met prefix `__strict-` verwacht wordt, maar deze niet werd meegestuurd omdat het ging om een cross-site request.



Figuur 4.2: Voorbeeld ter verduidelijking van het concept rond conversie.

Op deze manier slaagden we erin om de website volledig af te schermen van de implementatie van same-site cookies. De session cookie kan gebruikt worden zoals voordien, met een extra laag van beveiliging. De administrator zal enkel door middel van module-specifieke directieven moeten aangeven waar welke cookie geaccepteerd kan worden. Dit wordt besproken in de volgende sectie.

4.1.2 Configuratie van de module

Onze module biedt drie directieven aan om de module te configureren. Elk van deze directieven kan gebruikt worden op dezelfde manier als gewone Apache-directieven, door ze in de configuratiebestanden in te voegen.

Het eerste directief is `Enable_SS_Module` en heeft geen argument nodig. Dit directief geeft aan dat onze module ingeschakeld moet worden voor een bepaalde serverinstantie. Bij afwezigheid van het directief zal de module uitgeschakeld zijn. Dit directief dient vooral om in testen de module in en uit te kunnen schakelen.

Het `CookieName`-directief accepteert één argument en wordt ook gebruikt per serverinstantie. Dit argument bepaalt de naam van de cookie die elke keer geconverteerd zal moeten worden. De module wijzigt enkel de cookies die deze naam dragen in de `Cookie`- en `Set-Cookie`-headers. Voor deze cookies wordt enkel de cookie-

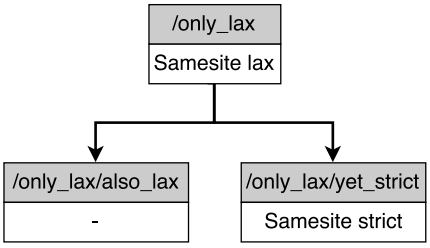
waarde gewijzigd, de attributen blijven hetzelfde. Alle andere cookies behouden hun oorspronkelijke vorm. De administrator kan dus via dit directief aanduiden welke session cookie opgesplitst zal worden, zonder dat dit effect heeft op andere cookies. In dit prototype kan er maar één cookienaam worden meegegeven via dit directief omdat dit genoeg is om het concept te testen en te evalueren. Het is niet moeilijk om de implementatie van het prototype te wijzigen zodat er meerdere cookies hiervoor kunnen worden opgegeven.

Het laatste directief heet `Samesite` en geeft aan welke cookie vereist is op welke plaats op de website. Dit directief accepteert één argument dat de waarde "gen", "lax" of "strict" mag hebben. Het directief wordt geplaatst in de context van <Directory>- en <Location>-tags. In deze context zal de module enkel de geconverteerde cookie accepteren wanneer deze de geassocieerde prefix draagt. Zo geven we de administrator de mogelijkheid om op een gemakkelijke manier een bepaalde strategie toe te passen voor het beveiligen tegen aanvallen gericht op de session cookie.

We illustreren het gebruik van dit directief aan de hand van een voorbeeld in listing 4.1. De module zal hier enkel `SameSite-lax` cookies converteren voor resources die zich bevinden in de map `only_lax` en enkel `SameSite-strict` cookies converteren voor resources die zich bevinden in de map `yet_strict`. Merk op dat de map `yet_strict` een submap is van `only_lax`. Submappen erven de configuratie van dit directief in een bovenstaande map over, tenzij dit overschreven wordt zoals in het geval van `yet_strict`. Op deze manier hoeft de administrator niet voor elke onderliggende map handmatig het directief te definiëren, enkel indien het gaat om een andere configuratie dan de bovenliggende map. Figuur 4.3 verduidelijkt dit door de gegevensstructuur weer te geven. De gegevensstructuur geeft in het grijze vakje de maplocatie aan en in het witte vakje het gebruikte `Samesite`-directief voor deze context. Het gaat hier om dezelfde configuratie uit listing 4.1 waarbij `only_lax` enkel `SameSite-lax` cookies accepteert. We hebben dit directief niet gedefinieerd voor de submap `also_lax`, dus als gevolg zal deze map ook enkel `SameSite-lax` cookies accepteren. Ook elke submap van `also_lax` zal deze configuratie overerven indien deze submap zelf deze configuratie niet overschrijft. Voor de map `yet_strict` gebruiken we daarentegen wel het `Samesite`-directief, namelijk met de intentie om de module hiervoor enkel `SameSite-strict` cookies te laten accepteren. Dit overschrijft de configuratie van de map `only_lax`.

LISTING 4.1: Een voorbeeld van het gebruik van het `Samesite`-directief

```
<Directory /var/www/html/only_lax>
        Samesite lax
</Directory>
<Directory /var/www/html/only_lax/yet_strict>
        Samesite strict
</Directory>
```



Figuur 4.3: Voorbeeld omtrent de overerving bij het `Samesite`-directief.

4.1.3 Contextuele cookieconversie

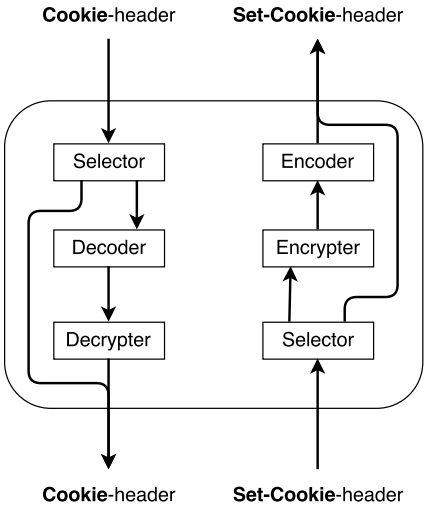
Zoals figuur 4.2 al aantoonde, wordt de aangeduide cookie telkens geconverteerd naar drie verschillende cookies en vice versa. Er moet echter ook rekening gehouden worden met contextuele informatie vooraleer de module deze conversie kan starten. Zo kan er immers pas voor gezorgd worden dat de geconverteerde cookies geaccepteerd worden op basis van de mogelijke impact van een actie. Om dit proces te verduidelijken, staat de stroom van de relevante headers afgebeeld in figuur 4.4.

We beginnen met de stroom van de `Cookie`-header. De module ontvangt deze header bij een request die wordt geïnitialiseerd door de webbrowser. Het is belangrijk dat de cookies die niet geconverteerd moeten worden strict gescheiden worden van de conversiefunctionaliteit. Hiervoor zorgt de `Selector`, die als eerste de header behandelt. Aan de hand van de cookienaam die door het `CookieName`-directief werd gespecifieerd, bepaalt de `Selector` welke cookies in deze header geconverteerd zullen worden en welke niet. De cookies die niet geconverteerd zullen worden, volgen de linkerpijl om samengevoegd te worden in hun originele staat met de geconverteerde cookie. Daarnaast heeft de `Selector` nog een andere taak. Er moet ook worden gezorgd dat de juiste cookie van het drietal geconverteerd wordt, afhankelijk van de context. Maximaal één cookie van dit drietal wordt dan doorgestuurd naar de `Decoder`, de andere twee worden verworpen. Dit is de cookie die geconverteerd mag worden volgens het `Samesite`-directief. Merk op dat het niet noodzakelijk is dat elke cookie van dit drietal aanwezig is in de `Cookie`-header. Indien de te converteren cookie niet aanwezig is in de header, zal de webserver geen gedecrypteerde session cookie ontvangen.

De `Decoder` decodeert dan op zijn beurt de cookie en stuurt deze door naar de `Decrypter`. De mogelijkheid bestaat dat een aanvaller geprobeerd heeft om een cookiewaarde te raden voor bijvoorbeeld de `SameSite-strict` cookie om zo toch een cross-site aanval uit te kunnen voeren op een impactvolle actie. Daarom checkt de `Decrypter` eerst of er niet geknoeid is met de waarde van de cookie en zal deze decrypteren indien dit niet het geval is. De cookie wordt verworpen indien ongeoorloofde modificatie wordt gedetecteerd. Uiteindelijk wordt de gedecrypteerde cookie (indien aanwezig) samen met de originele cookies afkomstig van de `Selector` doorgegeven in de header aan de Apache-server.

Nu zullen we de stroom van de `Set-Cookie`-header overlopen. Deze header is afkomstig van de Apache-server en bevat de cookies die opgeslagen zullen worden

door de webbrowser. Indien aanwezig zal de juiste cookie opgesplitst moeten worden in drie cookies, terwijl de andere cookies ongewijzigd doorgestuurd moeten worden. Dit is de taak van de `Selector` en ook hier zal het als eerste de header behandelen op basis van de cookienaam ingesteld met het `CookieName`-directief. De cookies die niet geconverteerd moeten worden, volgen de rechterpijl om samengevoegd te worden met de geconverteerde cookies. De cookie die geconverteerd moet worden, wordt doorgegeven aan `Encrypter`. Hier zal de cookie opgesplitst worden in drie cookies waarbij de cookiewaarde telkens geëncrypteerd wordt met een verschillende sleutel (`key_gen`, `key_lax` of `key_strict`). De sleutel wordt bepaald op basis van de prefix en het `SameSite`-attribuut dat de cookie zal krijgen. De geëncrypteerde cookiewaarden kunnen nog niet voorgesteld worden in leesbare symbolen. Daarom worden de drie geëncrypteerde cookies worden nog doorgegeven aan `Encoder` die hun cookiewaarden codeert in Base64-formaat. Het resultaat van de samenvoeging van de originele cookies en de geconverteerde cookies wordt dan opgestuurd in de `Set-Cookie`-header naar de webbrowser.



Figuur 4.4: Schematische voorstelling van de stroom van de headers in de module.

4.1.4 Encryptie en encodering

We hebben ervoor gekozen om dit probleem op te lossen via encryptie zodat we geen extra sessiegebonden informatie server-side moeten opslaan, terwijl de informatie toch voldoende beschermd blijft. Het prototype genereert de sleutels `key_gen`, `key_lax` en `key_strict` willekeurig bij het opstarten, alsook de respectievelijke initialisatievectors. Hiervoor is gekozen om het testen gemakkelijker te laten verlopen, maar in een echte toepassing is het de bedoeling dat deze sleutels opgeslagen worden zodat sessies ook na een reboot geldig blijven. Het is daarnaast ook belangrijk dat de sleutels in een real-world setting dynamisch vernieuwd kunnen worden, zonder dat gebruikerssessies

worden onderbroken. Dit is nodig voor het geval dat de sleutels achterhaald zijn of ter preventieve verversing na een bepaalde tijd van gebruik van dezelfde sleutels.

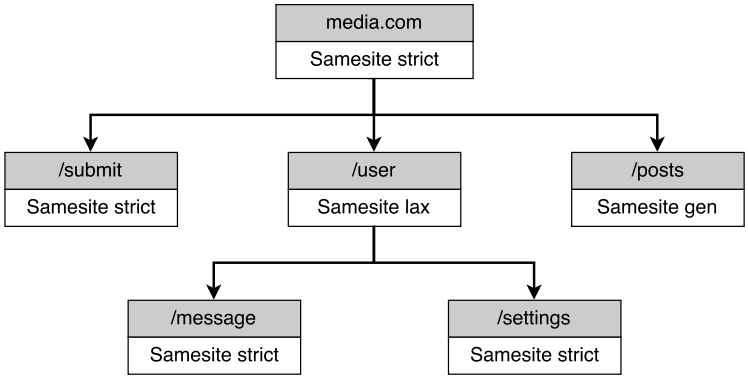
Voor de encryptie en codering van de cookies maken we gebruik van enkele OpenSSL-libraries (evp.h, buffer.h en rand.h). De encryptievorm moet er niet alleen voor zorgen dat de originele waarde van de cookie vertrouwelijk blijft, we willen eveneens dat de cookie beschermd is tegen mogelijke kwaadwillige aanpassingen. Daarom kozen we voor *Authenticated Encryption with Associated Data* (AEAD) in *Galois/Counter Mode* (GCM) als encyptievorm. Bij het ontvangen van een geconverteerde cookie kunnen we controleren of de cookie aangepast is (integriteit) en of de cookie wel werkelijk van onze webserver afkomstig is (authenticatie). De eigenlijke encryptie gebeurt door het AES-algoritme met een sleutel van 256 bits en een initialisatievector van 96 bits. Daarnaast telt de *Additional Authenticated Data* (AAD) 48 bits en de integriteitstag die op het einde wordt toegevoegd telt 128 bits. Voor de lengte van de sleutels, initialisatievectors, integriteitstag en AAD werd er telkens gebruik gemaakt van de defaultwaarden uit de OpenSLL-library. Encryptie aan de hand van deze lengtes is voldoende veilig, maar aanpassing is altijd mogelijk.

De geconverteerde cookies zijn wel langer dan de originele cookie. Stel dat x de lengte in bytes is van de originele cookie. Na encryptie van de originele cookie door middel van AES in GCM, is de lengte van het resultaat nog altijd x. Hier wordt echter wel nog een tag van 16 bytes aan toegevoegd, dit resulteert in een lengte van $x + 16$. De AAD hoeven we niet toe te voegen aan de cookie. Uiteindelijk wordt het resultaat geëncodeerd in Base64 zodat de cookie leesbaar is. Stel dat we een een string van lengte y encoderen, dan zal de Base64-geëncodeerde string per definitie een lengte hebben van $4 * \lceil \frac{y}{3} \rceil$. We nemen $y = x + 16$ en substitueren dit in de vorige formule, zodat we tenslotte een lengte van $4 * \left\lceil \frac{x+16}{3} \right\rceil$ bytes verkrijgen. De webbrowser zal dus drie cookies met een cookiewaarde van deze lengte moeten opslaan.

4.1.5 Voorbeeld use case

Om de werking en de adoptie van het prototype te verduidelijken, beschrijven we een voorbeeld waarin de module wordt toegepast. De website `media.com` gebruikt onze module om haar gebruikers meer bescherming te bieden tegen cross-site aanvallen. Dit social-mediaplatform geeft gebruikers de mogelijkheid om eigen content te plaatsen op de website in de vorm van een *post*. Op deze posts kan dan gereageerd worden door andere gebruikers. De website plaatst session cookies zodat gebruikers niet bij elke request handmatig in moeten loggen. De administrator koos voor de configuratie afgebeeld in figuur 4.5. Hierop is de gegevensstructuur van de website te zien samen met de geassocieerde configuratie van het `SameSite`-directief. De hiervoorgenoemde session cookie werd aangeduid als de te converteren cookie.

Op het hoogste niveau is er voor een strikte configuratie gekozen. Op deze manier wordt de strikte configuratie toegepast voor alle mappen zonder expliciete `SameSite`-configuratie, en dus ook voor de mappen die mogelijk over het hoofd gezien zouden kunnen zijn. Voor de rest heeft dit geen invloed op de werking van de website indien alle mappen correct geconfigureerd zijn.



FIGUUR 4.5: Voorbeeld van een configuratie voor een bepaalde use case.

De `submit`-map wordt gebruikt om gebruikers een post te laten insturen. Dit gebeurt altijd per authenticatie van de gebruiker die de post wil indienen. Deze functionaliteit kan niet geëmbed worden op andere domeinen en wordt beschouwd als impactvol. Omwille van deze redenen is er gekozen om hier enkel `SameSite-strict` cookies te accepteren voor authenticatie van een gebruiker. Zo wordt er een extra laag van bescherming gecreëerd tegen CSRF-aanvallen die bijvoorbeeld spam willen plaatsen in naam van geregistreerde gebruikers.

De `user`-map wordt gebruikt om de profielpagina van een gebruiker weer te geven. Deze pagina bevat algemene informatie over het gebruikersprofiel, maar ook informatie specifiek voor de bezoekende gebruiker. Zo staat er bijvoorbeeld aangegeven of dit profiel gemarkeerd staat als favoriet van de bezoekende gebruiker. Het gebruik van de `SameSite-strict` cookie is hier niet aangewezen. Een aanval in deze context zou niet van grote impact zijn omdat er weinig functionaliteit is om te misbruiken. Daarnaast zal een `SameSite-strict` cookie ervoor zorgen dat bij het doorklikken naar een profielpagina via een hyperlink op een andere website, de profielpagina getoond zal worden zonder de informatie specifiek voor de bezoekende gebruiker. De authenticerende cookie wordt dan immers niet meegestuurd bij een klik op de hyperlink. Voor dit geval is het dus handiger om een `SameSite-lax` cookie te gebruiken. De submap `message` wordt gebruikt om een bericht naar de gebruiker achter de profielpagina te sturen. Deze functionaliteit is dan wel weer impactvol omdat een aanvaller een bepaalde gebruiker zou kunnen viseren door met een CSRF-aanval andere gebruikers spam te laten sturen. De `SameSite-strict` cookie kan dit voorkomen. Omdat deze functionaliteit enkel vanaf `media.com` dient uitgevoerd te worden, wordt er ook niet ingeboet op vlak van gebruiksvriendelijkheid. Via de `settings`-map wordt toegelaten dat gebruikers elkaar als favoriet kunnen aanduiden, of elkaar kunnen blokkeren. Net zoals in het vorige geval, gaat het hier om een grotere impact indien hier misbruik van gemaakt kan worden via cross-site aanvallen. Daarom werd hier ook gekozen voor de `SameSite-strict` cookie.

Ten slotte is er nog de `posts`-map. Deze map verschaft toegang tot alle openbare posts die gemaakt zijn door gebruikers. Elke post bevat ook enkele knoppen die het de gebruikers mogelijk maakt om op een post te stemmen, op te slaan of te

rapporteren als ongewenst. Deze posts kunnen met inbegrip van deze knoppen geëmbed worden op andere domeinen, waardoor er gekozen is voor een generische cookie in de configuratie. Hoewel dit noodzakelijk is om deze functionaliteit aan te bieden, bestaan er wel risico's dat er CSRF-aanvallen worden uitgevoerd die gericht zijn op het stemmen, opslaan of rapporteren van posts in naam van een nietsvermoedende gebruiker. Deze risico's zouden op een andere manier aangepakt kunnen worden om het uiteindelijke doel van de website te vrijwaren.

4.2 Alternatieven

Naast de oplossing die we in dit hoofdstuk uitgebreid beschreven, hebben we ook enkele alternatieve oplossingen overwogen. Deze alternatieven bespreken we kort in deze sectie.

4.2.1 Server-side oplossing

In plaats van de drie verschillende cookies te laten volgen uit de encryptie van één oorspronkelijke cookie, zou de module ook gewoon drie verschillende extra session cookies kunnen opslaan per gebruikerssessie. Conform met onze implementatie zal er dan één generische, één `SameSite-lax` en één `SameSite-strict` cookie zijn. Op basis van impact bij een cross-site aanval kan hier ook weer beslist worden welke cookie vereist is op welk deel van de website.

De module zou voor de session cookie die ingesteld wordt met de `Set-Cookie`-header, drie unieke en onvoorspelbare cookies kunnen genereren. De oorspronkelijke cookie zal dan samen met de drie nieuwe cookies opgeslagen moeten worden. Bij ontvangst van een request zal de `Cookie`-header dan nagekeken worden en zal de oorspronkelijke cookie opgevraagd worden uit de verzameling van deze tupels indien de ontvangen gegenereerde cookie contextueel voldoet.

Een voordeel van deze aanpak is dat cookies niet iedere keer geconverteerd moeten worden en dat er geen manier is voor de aanvaller om de conversie van cookies te achterhalen. Hiervoor zal het genereren van de extra cookies evenwel voldoende veilig moeten gebeuren. Het grote nadeel is dan weer dat er server-side per gebruikerssessie extra informatie zal opgeslagen moeten worden, terwijl dit client-side ook nog altijd het geval is. De webbrowser slaat hier namelijk ook drie cookies op, zij het wel met een kortere cookiewaarde dan besproken in sectie 4.1.4. Deze oplossing schaalt ook veel moeilijker omdat een groeiend aantal gebruikerssessies zal zorgen voor een slechtere performantie, de te doorzoeken lijst van cookies wordt immers steeds groter. In tegenstelling tot deze oplossing, is de performantie van onze oplossing wel onafhankelijk van het aantal actieve gebruikerssessies.

4.2.2 Client-side oplossing

De ontbrekende ondersteuning door enkele grote webbrowsers zorgt ervoor dat websites nog niet volledig kunnen vertrouwen op in-depth bescherming door middel van same-site cookies. De browserkeuze van hun gebruikers is immers zeer divers,

waardoor er altijd wel een significant deel geen baat zou hebben bij de extra laag van bescherming. De ondersteunende webbrowsers Google Chrome en Opera bezitten samen namelijk maar net iets meer dan de helft van het wereldwijde marktaandeel van gebruikte webbrowsers volgens StatCounter [29]. Dit belemmert een spoedige adoptie, wat op zijn beurt dan weer weinig aansporing creëert voor de browserontwikkelaars om ondersteuning voor same-site cookies te implementeren. Om sneller uit deze vicieuze cirkel te geraken, dachten we na over een mogelijk client-side alternatief. Via dit alternatief zouden gebruikers van ondersteuningbiedende webbrowsers zelf kunnen kiezen voor een in-depth bescherming op basis van same-site cookies en hoeven ze niet te wachten op de logge adoptie van websites.

Een alternatief zou kunnen bestaan uit een browserextensie die via bepaalde policies `SameSite`-attributen toevoegt aan specifieke cookies. Deze policies definiëren is het belangrijkste gedeelte en kan op verschillende manieren gebeuren. Men zou de gebruikers zelf de mogelijkheid kunnen geven om te bepalen welke cookies same-site worden. Dit vereist echter een zeker niveau van kennis over het onderwerp en zal bijgevolg nog altijd een groot deel van gebruikers uitsluiten. Een verbetering zou zijn om - zoals bij CsFire [5] - een onderscheid te maken tussen lokale en remote policies, waarbij de remote policies worden opgesteld via een database die regelmatig wordt geüpdatet door experts.

Een andere interessante piste is om de policies op te stellen op basis van gebruikersgegevens. De extensie stuurt dan gegevens over gemaakte same-site en cross-site requests door naar een centrale server. Op basis van deze informatie kan dan per cookie beslist worden of deze een `SameSite`-attribuut krijgt of niet. Men zou kunnen beredeneren dat indien er veel cross-site requests geregistreerd worden naar een bepaalde endpoint van een website, deze niet gebeuren op basis van kwaadwillige motieven. Zo zal het plaatsen van een reactie op een geëmbedde Facebook-post voldoen aan dit criterium en beschouwd worden als een veilige cross-site request. Als gevolg zal de cookie die essentieel is voor deze actie niet same-site gemaakt worden omdat dit anders de functionaliteit van de website zou breken. Anderzijds zou registratie van veel same-site requests en weinig of geen cross-site requests naar een bepaalde endpoint beschouwd kunnen worden als een impactvolle actie. Het kan hier bijvoorbeeld gaan om het aanpassen van accountgegevens. De essentiële cookies voor deze requests kunnen dan same-site gemaakt worden. Aangezien er toch weinig cross-site requests gebeuren, zal dit bovendien weinig invloed hebben op de werking van de website.

Het komt echter voor dat de cookiestrategie van een website soms niet toelaat om bepaalde cookies same-site te maken, zonder bepaalde functionaliteit te breken. Vaak wordt immers dezelfde session cookie gebruikt om impactvolle én impactgeringe requests te authenticeren. Deze werkwijze is dus sterk afhankelijk van de cookiestrategie die gebruikt wordt per website. Er zal dan in veel gevallen een afweging gemaakt moeten worden tussen veiligheid en gebruiksvriendelijkheid. Omdat een server-side oplossing meer controle heeft over de cookies die ingesteld worden door de webserver, lijkt dit - ondanks te trage adoptie - nog altijd de meest geschikte oplossing.

Hoofdstuk 5

Evaluatie van het prototype

In dit hoofdstuk zullen we het ontwikkelde prototype evalueren op basis van performantie (sectie 5.1) en veiligheid (sectie 5.2).

5.1 Performantie

Het is belangrijk dat het prototype geen significante verhoging van de last veroorzaakt voor een webserver, daarom evalueren we in deze sectie de performantie. Dit doen we op basis van de benodigde rekentijd die de CPU nodig heeft bij het uitvoeren van de module voor elke request. We hebben deze performantietest gedaan op basis van de rekentijd gemeten bij het converteren van de geëncrypteerde cookies naar de oorspronkelijke cookie, omdat deze conversie het vaakst zal voorkomen. Dit is de conversie van de `Cookie`-header zoals afgebeeld in figuur 4.4. Voor deze test zorgen we eerst dat alle relevante cookies (de `__gen-`, de `__lax-` en de `__strict-` cookie) ingesteld zijn door de webbrowser. Daarna bezoeken we om de beurt een pagina die telkens één van deze cookies converteert naar de originele cookie. Deze pagina's zijn respectievelijk `/only_gen`, `/only_lax` en `/only_strict`. Dit proces herhalen we 40 keer, waardoor de webserver in totaal 120 conversies zal hebben uitgevoerd. In listing 5.1 wordt dit nog eens beschreven met behulp van pseudocode. Alle requests werden gelogd en getimed door de webserver. De resultaten hiervan bespreken we dan aan de hand van een boxplot.

Listing 5.1: De pseudocode ter evaluatie van de performantie.

```
Sla de geconverteerde ___gen−, ___lax− en ___strict−cookies op;
for i = 1..40:
    Bezoek /only_gen;       // De module logt de benodigde
    Bezoek /only_lax;       // rekentijd voor de conversie
    Bezoek /only_strict;    // bij elk bezoek.
```

5.1.1 Performantie bij gebruik van een standaard cookie

We hebben onderzocht wat de benodigde rekentijd van de module is bij gebruik van een cookiewaarde van 40 bytes. Deze resultaten staan afgebeeld in de boxplot `Test 1` van figuur 5.1. De benodigde extra rekentijd ten gevolge van de module is gemiddeld 25,6 µs. Het valt echter op dat er zich enkele extreme uitschieters bevinden rond de 100 µs. De gelabelde uitschieters a, b en c zijn de allereerste drie requests met respectievelijk de conversie van de `__gen-`, `__lax-` en `__strict-` cookie. We vermoeden dat de hieropvolgende requests beïnvloed worden door caching waardoor de tijdsmetingen veel gunstiger ogen. De encryptiesleutels, oorspronkelijke cookiewaarde en/of de geconverteerde cookiewaarden bevinden zich mogelijk nog altijd in RAM-geheugen bij de volgende conversies.

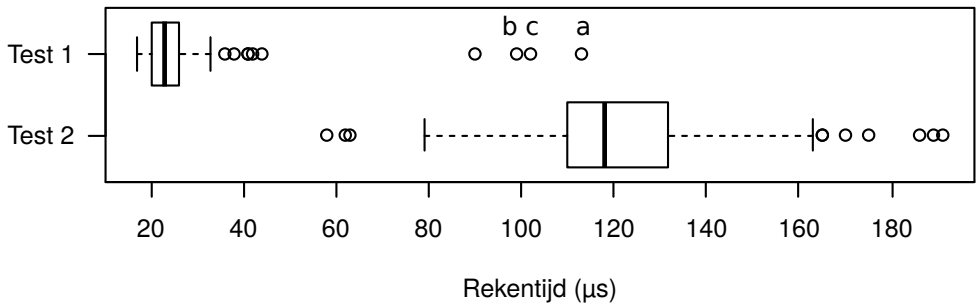
Hoewel deze vorm van caching (in beperkte mate) ook kan toegepast worden bij echte webservers, willen we deze test toch ook uitvoeren in een worst-case scenario waarbij caching wordt uitgesloten. In deze nieuwe test starten we, in tegenstelling tot de vorige test, de webserver iedere keer opnieuw op wanneer we de drie verschillende cookies geconverteerd hebben. Dit zorgt ervoor dat de encryptiesleutels telkens opnieuw gegenereerd worden en de geconverteerde cookiewaarden verschillend zullen zijn. Voor elke herhaling van het proces (40 keer), worden er dus telkens nieuwe encryptiesleutels gegenereerd. De gewijzigde pseudocode voor de evaluatie wordt beschreven in listing 5.2. Figuur 5.1 geeft de resultaten van onze tweede test weer met de boxplot `Test 2`. We merken meteen op dat de uitschieters minder extreem zijn in vergelijking met de vorige test. Daarnaast is het gemiddelde significant gestegen naar 122,3 µs.

Listing 5.2: De gewijzigde pseudocode ter evaluatie van de performantie in een worst-case scenario.

```
for i = 1..40:
    Herstart de webserver;
    Sla de geconverteerde ___gen−, ___lax− en ___strict−cookies op;
    Bezoek /only_gen;       // De module logt de benodigde
    Bezoek /only_lax;       // rekentijd voor de conversie
    Bezoek /only_strict;    // bij elk bezoek.
```

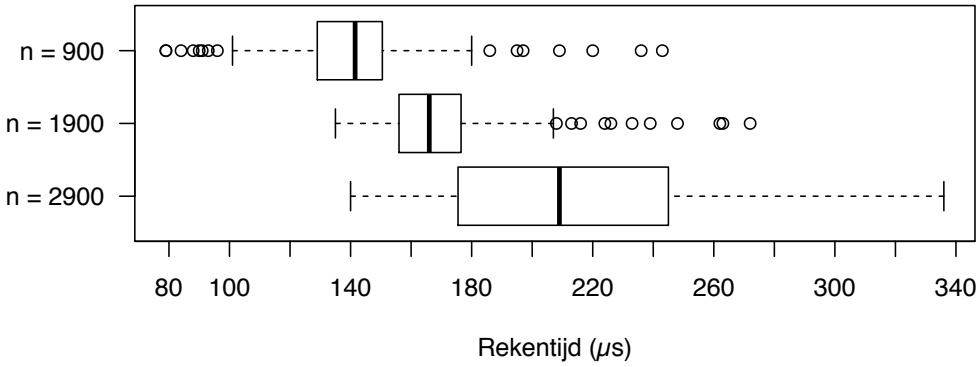
5.1.2 Invloed van de cookiegrootte op de performantie

We onderzochten ook of de grootte van de oorspronkelijke cookiewaarde een significante invloed zou hebben op de performantie. Hiervoor hielden we rekening met de maximale cookiegrootte die ondersteund wordt door de meeste webbrowsers, namelijk 4096 bytes. Dit is inclusief de cookienaam en cookie-attributen. De module vergroot de oorspronkelijke cookiewaarde zoals besproken in sectie 4.1.4. Met het oog op de vergroting van de cookiewaarde en de grootte van de cookienaam en cookie-attributen, kozen we voor een maximale cookiewaarde van 2900 bytes in deze test. Hierbij moet wel vermeld worden dat het noodzakelijk was de default limiet voor request headers van de Apache-server te verhogen met behulp van het



FIGUUR 5.1: De resultaten van de performantietest voor een cookiewaarde van 40 bytes. Voor `Test 1` werd er caching gebruikt, bij `Test 2` is dit niet het geval.

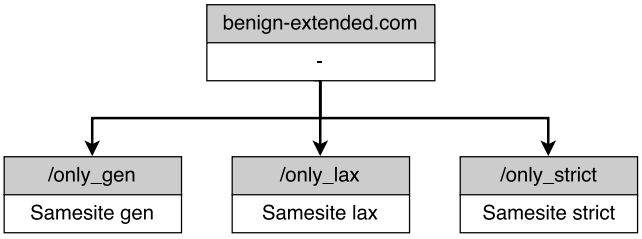
`LimitRequestFieldSize`-directief. We evalueren het worst-case scenario waarbij de server elke keer opnieuw wordt opgestart zoals in listing 5.2. We benoemen n als de grootte van de oorspronkelijke cookiewaarde en meten de benodigde rekentijd voor n = 900, 1900 en 2900 bytes. De resultaten van de test staan afgebeeld in figuur 5.2. De gemiddeldes zijn respectievelijk 140,6 µs, 171,2 µs en 212,5 µs. Hieruit kunnen we afleiden dat de grootte van de cookiewaarde wel degelijk een invloed heeft op de benodigde rekentijd voor de module. De rekentijd blijft echter nog altijd van dezelfde grootteorde.



FIGUUR 5.2: De resultaten van de performantietest voor cookiewaarden van 900, 1900 en 2900 bytes.

5.2 Kwetsbaarheden

We hebben onze module ook getest op bestaande aanvallen. Voor veel van de volgende evaluaties maken we gebruik van de setup die wordt afgebeeld in figuur 5.3. We beschikken dus over een website `benign-extended.com` waarvan de webserver is uitgebreid met de module die we in hoofdstuk 4 hebben beschreven. Deze setup laat ons toe om te achterhalen welke cookies vatbaar zijn voor bepaalde aanvallen. Ter referentie wordt `/only_gen` gebruikt, waar enkel de generische cookie wordt geaccepteerd. De andere locaties, `/only_lax` en `/only_strict`, accepteren respectievelijk enkel de `SameSite-lax` en enkel de `SameSite-strict` cookie. Bij een bezoek aan `https://benign-extended.com` zal de webserver de cookie `sid` met een bepaalde waarde en bijhorende attributen instellen. Deze cookie wordt telkens door onze module geconverteerd. Het resultaat is de drie geconverteerde cookies die ingesteld worden via de `Set-Cookie`-header zoals getoond wordt in listing 5.3. De webbrowser slaat dus de drie besproken geconverteerde cookies op die gebruikt worden om een gebruikerssessie the identificeren. Naast same-site cookies is er geen ander beveiligingsmaatregel getroffen ter bescherming van de evaluatie-setup. Deze testen zijn uitgevoerd met Chromium.



Figuur 5.3: Opstelling van de evaluatie-omgeving.

Listing 5.3: De cookies die ingesteld worden door de webbrowser.

```
Set−Cookie: ___gen−sid=hLvh72Nk...; domain=benign−extended.com; path=/
Set−Cookie: ___lax−sid=DG06H0oU...; domain=benign−extended.com; path=/; SameSite=lax
Set−Cookie: ___strict−sid=oyFnUV...; domain=benign−extended.com; path=/; SameSite=strict
```

5.2.1 CSRF

In deze sectie bespreken we de verschillende attack vectors die gebruikt kunnen worden voor CSRF en in hoeverre same-site cookies hiertegen bestand zijn. We gaan ervan uit dat staatswijzigende acties doorgevoerd kunnen worden door zowel GET- als POST-requests.

Clickjacking

Zoals we beschreven hebben in sectie 3.3.1, wordt de `SameSite-lax` cookie meegestuurd bij een cross-site request als gevolg van een klik op een hyperlink. Hiervoor

is echter niet noodzakelijk een hyperlink nodig, tabel A.1 toont nog verschillende manieren om via interactie met de gebruiker toch een `SameSite-lax` cookie te versturen. Dit kan de aanvaller inspireren om via clickjacking de gebruiker op een bepaald HTML-element te laten klikken die een CSRF-aanval in gang zet. De aanvaller zou dit bijvoorbeeld kunnen uitvoeren via een advertentie. De grote zichtbaarheid van deze aanval werkt evenwel in het nadeel van de aanvaller. Om een `SameSite-lax` cookie door de sturen, is er immers een top-level navigatie nodig die heel zichtbaar is voor de gebruiker (met uitzondering van het prerender-element). Voor de evaluatie op basis van de omgeving weergegeven in figuur 5.3 verkregen we voorspelbare resultaten voor clickjacking. Enkel voor `/only_strict` werd de CSRF-aanval afgeweerd, nadat de gebruiker op een kwaadwillige hyperlink geklikt had.

Populaire websites bieden vaak zelf embeddable functionaliteit aan. Facebook laat bijvoorbeeld toe om vanaf een andere website reacties en likes te plaatsen op een bepaalde Facebook-post of -pagina. Door deze functionaliteit te embedden op zijn eigen website, kan de aanvaller het slachtoffer hier ook mee laten interageren door middel van clickjacking. Deze manier van clickjacking wordt besproken in het onderzoek van Huang et al. [14]. Aangezien deze functionaliteit cross-site requests verstuurt die geen top-level navigatie veroorzaken, wordt er geen enkele same-site cookie meegestuurd. Deze vorm van clickjacking kan dus voorkomen worden door middel van same-site cookies. We moeten wel opmerken dat indien er een top-level navigatie gebeurt ten gevolge van een interactie met deze embeddable content, er toch een `SameSite-lax` cookie kan worden meegestuurd met een cross-site GET-request.

Voor websites die geen embeddable functionaliteit aanbieden, kan de aanvaller dit toch forceren door gebruik te maken van iframes. Door middel van een iframe kan de aanvaller bijvoorbeeld een pagina met een in te vullen formulier van de doelwebsite op zijn eigen website plaatsen. Clickjacking door middel van iframes werd onderzocht door Rydstedt et al. [27]. Men analyseerde hier bestaande *frame busting*-technieken die gebruikt worden door populaire websites. Hieruit volgde dat de grote meerderheid van websites uit de Alexa Top-500 geen adequate *frame busting*-technieken gebruiken. We evalueerden de werking van de same-site cookie in deze context door acties uit te voeren binnen een iframe. Voor deze acties, (bv. het klikken op een hyperlink, het indienen van een form) werd telkens geen enkele same-site cookie meegestuurd met de request die geïnitieerd werd door de actie. Same-site cookies lijken dus een goed alternatief voor *frame busting*.

Er bestaan al verschillende manieren waarop clickjacking aangepakt kan worden. Enerzijds kunnen de webadministrators zelf maatregelen nemen door extra controles te verrichten bij bepaalde cross-site requests. Zo kan er om een extra verificatie worden gevraagd wanneer de cross-site request afkomstig is vanaf een verdacht domein. Het inbouwen van een willekeurigheid in de gebruikersinterface van een formulier is een meer doelgerichte manier om het de clickjacker moeilijk te maken. Op deze manier is het moeilijker om een bepaalde knop bovenop de legitieme knop te plaatsen wanneer de aanvaller dit formulier op zijn eigen pagina invoegt. Anderzijds kan er ook gebruik gemaakt worden van een client-side beveiliging. De webbrowser Gazelle pakt dit aan met zijn *opaque overlay policy* [33]. Deze policy zorgt ervoor dat ingevoegde cross-site elementen niet transparant kunnen zijn, zodat er geen onzichtbare elementen

gebruikt kunnen worden voor clickjacking. Een andere manier is om een beveiliging te gebruiken in de webbrowser die clicks niet doorvoert wanneer het element niet volledig zichtbaar is. Zoals bij de meeste beveiligingsmechanismen zal men hier ook weer een afweging moeten maken tussen veiligheid en gebruiksvriendelijkheid.

We zagen dat clickjacking door middel van hyperlinks nog altijd cross-site requests veroorzaakt met `SameSite-lax` cookies. Buiten het nemen van maatregelen tegen clickjacking in het bijzonder, zou men ook kunnen opteren om `SameSite-strict` cookies te gebruiken. De ontwikkelde module laat toe om impactsgewijs bepaalde functionaliteit van een website te laten afhangen van een strenger cookiebeleid zonder andere functionaliteit in gedrang te brengen. Zo kan het uitvoeren van een betaling bijvoorbeeld enkel geauthenticeerd worden door de `__strict`-cookie terwijl het plaatsen van een reactie geauthenticeerd kan worden door een `__lax`-cookie. Een doordacht cookiebeleid kan dus zeker een degelijke verdediging bieden voor impactvolle acties tegen clickjacking. Ten slotte willen we nog even de inconsistentie beschreven in sectie 3.3.4 aanhalen. Hier beschreven we dat het openen van een hyperlink in een nieuw tabblad of venster er wel voor zorgde dat een `SameSite-strict` cookie wordt meegestuurd met een cross-site request. Als de aanvaller een gebruiker zou kunnen overtuigen om de link op deze manier te openen, is de website alsnog kwetsbaar voor een CSRF-aanval.

Automatisch refereren

De aanvaller kan naast clickjacking ook elementen op zijn webpagina invoegen die de browser automatisch een cross-site request laten verzenden. In sectie 3.3.1 hebben we onderzocht welke same-site cookies bij welke automatische requests werden meegestuurd. We overlopen de meest voorname elementen en enkele methoden die niet betrokken werden in die analyse.

De **<img>- en <iframe>-tag** zijn veelgebruikte HTML-elementen voor CSRF-aanvallen. De webbrowser wordt als het ware bedot door de aanvaller om een bepaalde resource op te halen afkomstige van een andere website. Bij sommige websites veroorzaakt dit daarentegen de doorvoering van een bepaalde actie op de webserver. Volgens de analyse in sectie 3.3.1 wordt er geen enkele same-site cookie meegestuurd bij cross-site requests geïnitieerd door deze elementen. Het komt dan ook niet als verrassing dat bij evaluatie van beide methoden enkel de aanval op `/only_gen` succesvol was. De same-site cookies werden dus zonder probleem uitgesloten van de cross-site requests. Omdat onzichtbare iframes effectieve en verborgen CSRF-aanvallen mogelijk maken, zijn same-site cookies zeker geschikt om hier een extra laag van bescherming tegen te bieden. Jammer genoeg zijn er enkele sterke alternatieven die de aanvaller ter vervanging kan gebruiken. Deze komen verder in deze sectie aan bod.

Redirects kunnen ook gebruikt worden in een CSRF-aanval waarbij geen interactie van de gebruiker nodig is. We hebben het hier over server-side redirects door middel van de `30x` HTTP-statuscode, maar ook over client-side redirects. Op basis van een redirect kan de aanvaller de webbrowser doorverwijzen naar een andere pagina, waardoor de webbrowser automatisch een nieuwe request zal versturen naar

de gedefinieerde URL. Voor cross-site requests via redirects wordt de `SameSite-lax` cookie inbegrepen. Het grote nadeel hier is echter dat de aanval zeer zichtbaar is voor de gebruiker, de webbrowser toont namelijk de pagina op welke de aanval gericht is. De HTML-code uit listing 3.3 en 3.4 kan grotendeels overgenomen worden voor een aanval op `benign-extended.com`, enkel de URL zal gewijzigd moeten worden. Na evaluatie van beide methoden concluderen we dat naast een aanval op `/only_gen`, een aanval op `/only_lax` inderdaad ook succesvol zou zijn. Dit is niet het geval voor `/only_strict`.

Net zoals redirects kunnen **<form>-elementen** gebruikt worden voor een CSRF-aanval van hetzelfde kaliber. Door middel van een client-side script kunnen de forms automatisch ingediend worden, waardoor er hier ook geen interactie van de gebruiker vereist is. Het indienen zal echter wel door middel van een GET-request moeten gebeuren, daar same-site cookies uitgesloten worden bij POST-requests. Ook hier zal bij indiening een redirect gebeuren wat de aanval zichtbaar maakt voor de gebruiker. Hoewel aanvallen op basis van client-side redirects en <form>-tags nogal opvallend zijn, vormen ze wel een mogelijk alternatief voor de aanvallen die onschadelijk zijn gemaakt door same-site cookies.

CSRF-aanvallen kunnen ook geïnitieerd worden door middel van **XmlHttpRequest** (XHR) [38]. XHR staat toe een GET-, POST-, HEAD-, PUT-, DELETE- of OPTIONS-request te versturen naar een bepaalde URL. Enkel de GET-, POST- en HEAD-requests hiervan zijn te gebruiken voor een CSRF-aanval omdat de rest toestemming nodig heeft via CORS. In de simulatie slaagden we er niet in om een same-site cookie mee te sturen met deze cross-site requests. Enkel `/only_gen` was dus kwetsbaar voor een CSRF-aanval via XHR. De experimentele Fetch API [19] is een alternatief voor XHR. Ook hier testten we de GET-, POST- en HEAD-request, omdat deze geen CORS-interactie vereisen. De resultaten waren identiek aan die van XHR, de aanval had enkel succes op `/only_gen`.

In sectie 3.3.1 bespraken we ook de **prerender-bug**. Het prerender-element zorgt ervoor dat een specifieke pagina achter de schermen vroegtijdig wordt ingeladen door de webbrowser zodat deze zonder vertraging getoond kan worden indien de gebruiker de pagina opvraagt [12]. Volgens de same-site cookies Internet Draft zou deze enkel een cross-site request kunnen veroorzaken waarbij de `SameSite-strict` cookies worden uitgesloten. We hebben ontdekt dat dit niet het geval is en de webbrowser elke cookie meestuurt, zonder rekening te houden met het `SameSite`-attribuut. De evaluatie bevestigt deze inconsistentie, een gesimuleerde aanval door middel van een prerender-element is succesvol voor elke map van `benign-extended.com`. Dit is een zeer ernstig probleem omdat er van een `SameSite-strict` cookie net verwacht wordt dat het met geen enkele cross-site request meegestuurd kan worden. We laten deze bug buiten beschouwing in deze evaluatie, het probleem is immers erkend en zal hoogstwaarschijnlijk snel opgelost worden.

Indien same-site cookies zouden werken zoals beschreven in de Internet Draft, worden `SameSite-lax` cookies nog altijd meegestuurd met cross-site requests geïnitieerd door het prerender-element. Voor een CSRF-aanval is dit een effectief alternatief ten opzichte van de andere minder geschikte methoden besproken in deze sectie. De aanval is nagenoeg onzichtbaar, wordt automatisch uitgevoerd door de webbrowser

en elke actie die een `SameSite-lax` cookie in een GET-request vereist kan gebruikt worden als doelwit. Het prerender-element wordt niet door elke webbrowser ondersteund, maar wel door elke webbrowser die op moment van schrijven same-site cookies ondersteunt. Het is wel bekend dat men de intentie heeft om de prerender-functionaliteit te verwijderen, er wordt namelijk al een alternatief ontwikkeld dat dezelfde performantiedoelen zou waarborgen [13]

Login CSRF

Login CSRF is een variant van de CSRF-aanval. In plaats van acties te verrichten met het account van het slachtoffer, probeert men hier het slachtoffer acties te laten verrichten met een account van de aanvaller. Omdat de aanvaller toegang heeft tot zijn eigen account, kan hij achteraf nakijken welke acties er gebeurd zijn door het slachtoffer. Het laten inloggen van een slachtoffer op het kwaadwillige account gebeurt op dezelfde manier als een gewone CSRF-aanval. Het slachtoffer bezoekt de website van de aanvaller, waarop de browser een cross-site request zal versturen die de login mogelijk maakt. Deze login zal een session cookie instellen op de webbrowser van het slachtoffer. Indien het slachtoffer al ingelogd was op de website, zal zijn session cookie overschreven worden. Uiteindelijk lijkt het alsof het slachtoffer (nog altijd) toegang heeft tot zijn eigen account.

Same-site cookies, noch onze module bieden voldoende bescherming tegen deze aanval. Bij het simuleren van deze aanval werden alle cookies, met inbegrip van de `SameSite-lax` en `SameSite-strict` cookies, voor `benign-extended.com` ingesteld wanneer het slachtoffer de webpagina van de aanvaller bezoekt. Hierna komen er geen cross-site requests meer aan te pas omdat het slachtoffer zelf zijn acties uitvoert via same-site requests. Als gevolg worden alle cookies met deze requests meegestuurd. De aanvaller is dan in staat om achteraf wijzigingen of logs van het account na te kijken.

Om dit probleem op te lossen, zou men kunnen opteren om het `SameSite`-attribuut niet enkel de machtiging te geven over het meesturen van de cookie, maar ook over het instellen van de cookie. Om in dezelfde lijn te blijven met de bestaande regels, zou een `SameSite-lax` cookie bijvoorbeeld enkel ingesteld kunnen worden met een cross-site request die aan dezelfde voorwaarden voldoet als gedefinieerd voor het meesturen van de cookie. Een `SameSite-strict` cookie zou dan helemaal niet ingesteld kunnen worden door middel van een response op een cross-site request. Anderzijds zal het dan natuurlijk ook niet meer mogelijk zijn voor legitieme websites om op deze manier same-site cookies in te stellen.

Gelukkig zijn er al verschillende maatregelen beschikbaar die op een andere manier bescherming kunnen bieden tegen Login CSRF. Zoals we al bespraken in sectie 2.3.2 kunnen CSRF-tokens en `Referer`-header validatie voldoende bescherming bieden, mits correct toegepast.

5.2.2 XSSI

In hoofdstuk 2 bespraken we enkele varianten van de XSSI-aanval. Bij elke variant komt het echter op hetzelfde neer: via een geauthenticeerde cross-site request wordt getracht gevoelige informatie aanwezig op de webserver te lekken. Deze informatie kan gelekt worden doordat de website onveilige methoden gebruikt om cross-site informatie te delen (JSONP, dynamische JavaScript [18]) of door bugs in de webbrowser uit te buiten (identifier based XSSI [31]). De aanvallen besproken in onderzoek van Lekies et al. en Terada worden uitgevoerd door middel van de <script>-tag [18, 31]. We weten al dankzij de analyse uit sectie 3.3.1, dat same-site cookies niet worden meegestuurd met cross-site requests geïnitieerd door de <script>-tag, zodat deze requests niet geauthenticeerd kunnen worden door middel van een same-site cookie.

Dit wordt ook bevestigd in de evaluatie van deze aanval op same-site cookies. Zowel de `SameSite-lax` als de `SameSite-strict` cookie worden niet meegestuurd met de cross-site requests. Hier is de aanvaller dus enkel in staat om gevoelige informatie te lekken aanwezig in `/only_gen`.

5.2.3 Timing attacks

Bortz en Boneh verklaren dat een timing attack door middel van een <img>-tag het meest effectief was in hun onderzoek, maar vermelden ook gebruik van iframes hiervoor [4]. In de analyse van sectie 3.3.1 zagen we al dat er geen same-site cookies worden meegestuurd met cross-site requests geïnitieerd door deze elementen. Op deze wijze kan dus alleen `/only_gen` aangevallen worden, terwijl `/only_lax` en `/only_strict` voldoende bescherming krijgen op basis van same-site cookies.

Omdat het prerender-element wel cross-site requests geauthenticeerd door same-site cookies kan initiëren, hebben we getracht deze aanval op een gelijkaardig manier te simuleren door middel van het prerender-element. Het onhandige hier is echter dat het inladen van een webpagina via een prerender-element geen foutmelding veroorzaakt, waardoor we het `onerror`-event niet kunnen gebruiken voor de stopzetting van de tijdsmeting. Hier komt nog eens bij dat het `onload`-event niet wordt afgevuurd bij een prerender-element. Daarnaast wordt enkel het eerste prerender-element in het HTML-document daadwerkelijk uitgevoerd, omdat er per HTML-document maar één webpagina via het prerender-element preventief ingeladen kan worden. Het is dus moeilijker om verschillende metingen te doen. Vanwege deze problemen hebben we geen effectieve cross-site timing attack kunnen uitvoeren op basis van het prerender-element.

Bortz en Boneh beschrijven ook nog een manier waarop timing attacks uitgevoerd kunnen worden wanneer er geen gebruik gemaakt kan worden van JavaScript. Zoals in het voorbeeld van listing 2.3 kan een aanvaller dan toch nog de benodigde tijd bepalen. We hebben geprobeerd deze aanval te simuleren met de <script>-tag, maar ook met het prerender-element. Hoewel anders beschreven in de paper van Bortz en Boneh, werden tags in onze test al behandeld voordat de vorige tag volledig was verwerkt door de webbrowser. Dit is te wijten aan het feit dat de resources sindsdien asynchroon ingeladen worden. Hierdoor konden we de tijd de nodig is om

een resource op te halen niet correct meten op deze manier. In onze evaluatie kon deze attack vector dus niet succesvol gebruikt worden.

Het onderzoek van Van Goethem et al. over browser-based timing attacks vermeldt verschillende attack vectors. Hiervan zijn de <audio>-, <video>- en <script>-tags er enkele. Geen van deze elementen veroorzaakt een cross-site request waarbij same-site cookies worden meegestuurd. Dit volgde al uit onze analyse in sectie 3.3.1 en bijhorende tabel A.1. Bijgevolg slaagden deze aanvallen enkel voor `/only_gen`. De andere attack vectors maken gebruik van ApplicationCache [22] en Service Workers [25]. Omdat ApplicationCache uiteindelijk helemaal vervangen zal worden door Service Workers evalueerden we enkel deze laatste. Via de Service Workers werd in het onderzoek van Van Goethem et al. de Fetch API [19] gebruikt voor de timing attacks. Bij het uitvoeren van een aanval via de Fetch API werd geen enkele same-site cookie meegestuurd met cross-site requests. We kunnen dus besluiten dat browser-based timing attacks enkel `/only_strict` kunnen kiezen als effectief doelwit in onze evaluatie.

Size-exposing attack

In hun onderzoek over size-exposing attacks gebruiken Van Goethem at al. ook de Fetch API om cross-site resources op te slaan [11]. Zoals we hiervoor al zagen, worden same-site cookies niet meegestuurd met cross-site requests geïnitieerd door methoden uit de Fetch API. Ten gevolge hiervan, kon deze aanval enkel succesvol uitgevoerd worden op `/only_strict`.

5.2.4 Technieken om same-site cookies te omzeilen

In deze sectie bespreken we methoden om same-site cookies te omzeilen. Op deze manier kunnen aanvallen die onschadelijk gemaakt worden door gebruik van same-site cookies toch succesvol uitgevoerd worden.

Uitbuiting van subdomeinen

Zoals besproken in sectie 3.3.3, worden alle same-site cookies meegestuurd met cross-site requests tussen subdomeinen. Zelfs tussen sub- en superdomein wordt er geen onderscheid gemaakt voor same-site cookies. Dit is een aantrekkelijk gegeven voor de aanvaller, daar hij same-site cookies niet alleen kan omzeilen door code te injecteren op de doelwebsite, maar ook op de sub- en superdomeinen van de doelwebsite. Met andere woorden is de attack surface om same-site cookies te omzeilen veel groter dan de doelwebsite alleen en bevat deze ook zijn sub- en superdomeinen.

Om dit te illustreren, breiden we de evaluatie-setup uit met twee subdomeinen: `sub1.benign-extended.com` en `sub2.benign-extended.com`. Via deze uitbreiding tonen we aan dat door middel van code injectie op eender welk sub- of superdomein de bescherming door same-site cookies omzeilt kan worden voor een bepaalde website. Stel dat een aanvaller de gebruikers van `benign-extended.com` als doelwit uitkiest. Deze aanvaller weet dat het domein gebruik maakt van same-site cookies en probeert via de subdomeinen toch een CSRF-aanval uit te voeren.

Dit kan op verschillende manieren gebeuren afhankelijk van de situatie. Indien `benign-extended.com` een *Content Delivery Networks* (CDN) is en zijn subdomeinen verhuurt aan klanten, kan de aanvaller op legale wijze controle verkrijgen over een subdomein door het te huren. Hij kan echter ook uitzoeken welke subdomeinen kwetsbaar zijn voor XSS-aanvallen en dit als luik gebruiken naar een CSRF-aanval tegen `benign-extended.com`. We hebben dit gesimuleerd aan de hand van een onzichtbare iframe die geïnjecteerd werd op `sub1.benign-extended.com`. Bij een bezoek aan deze pagina werd er automatisch een cross-site request verstuurd die alle drie de cookies uit listing 5.3 bevatte. Hetzelfde resultaat kan bereikt worden door code te injecteren op `sub2.benign-extended.com`

De aanvaller kan het echter ook gemunt hebben op het subdomein `sub1.benign-extended.com`. Hij kan er dan voor kiezen om via `benign-extended.com` of `sub2.benign-extended.com` de same-site cookies te omzeilen. Slechts één van deze domeinen hoeft hiervoor kwetsbaar te zijn voor XSS-aanvallen, of de aanvaller zou de mogelijkheid kunnen hebben om op legale wijze controle te verkrijgen over `sub2.benign-extended.com`. We hebben voor beide domeinen een CSRF-aanval gesimuleerd op `sub1.benign-extended.com` door middel van injectie van een onzichtbare iframe en ook hier werd elke geconverteerde cookie meegestuurd. Op dezelfde manier kan `sub2.benign-extended.com` aangevallen worden.

Cookie tampering

Cookie tampering staat voor het kwaadwillig aanpassen of creëren van een cookie op de webbrowser van het slachtoffer met als doel om toegang te verkrijgen tot de sessie van het slachtoffer. In deze sectie onderzoeken we methoden waarop een same-site cookie gemanipuleerd kan worden op deze manier. We willen echter niet de sessie van het slachtoffer stelen, maar wel het `SameSite`-attribuut van zijn same-site cookies verwijderen. Zo zullen deze cookies, na deze aanval, wel meegestuurd worden met alle cross-site requests, waardoor de same-site functionaliteit omzeild wordt. Deze aanval wordt verricht met behulp van XSS. Hiervoor hoeft de aanvaller niet noodzakelijk code te injecteren op het eigenlijke domein van de website. Zoals we in sectie 3.3.3 al aanhaalden, hebben subdomeinen toegang tot cookies van het hoofddomein. Via deze weg kunnen cookies ook gemanipuleerd worden, zoals al aangetoond werd door Zheng et al. [41]. We maken hier weer gebruik van de testomgeving die staat afgebeeld in figuur 5.3.

Cookie overwriting zorgt ervoor dat een bepaalde cookie van het slachtoffer overschreven wordt. Dit kan gedaan worden door gewoon een nieuwe cookie in te stellen met dezelfde naam, `Path`- en `Domain`-attributen door middel van een client-side script. De waarde van de cookie kan op deze manier aangepast worden, alsook de andere attributen. Bij de evaluatie proberen we het `SameSite`-attribuut aan te passen om zo de werking van de same-site cookie en onze module te omzeilen.

We gaan ervan uit dat de aanvaller in staat is om ergens op het domein `benign-extended.com` - of op een van zijn subdomeinen - een script te injecteren. Hij weet ook hoe de gebruikte session cookie heet en wat de waarden van het

`Path`- en `Domain`-attribuut zijn. Op basis van het client-side script kan hij dan de waarde van deze session cookie opslaan in een nieuwe cookie met exact dezelfde attributen, enkel het `SameSite`-attribuut laat hij vallen. Op deze wijze slaagden we erin om het de same-site cookies te transformeren in generische cookies zonder `SameSite`-attribuut.

Same-site cookies kunnen ook omzeild worden door middel van **cookie shadowing**. Hier wordt de cookie niet overschreven, maar wordt er een nieuwe cookie ingesteld die een hogere prioriteit heeft in vergelijking met de originele cookie. De webbrowser zal dus eerder de nieuwe cookie meesturen dan de originele cookie. Dit is volledig afhankelijk van hoe webbrowsers en webservers cookies met dezelfde naam behandelen, al wordt er in RFC6265 toch aangeraden om dit op een specifieke manier te doen [1]. Hierin wordt namelijk gestipuleerd dat cookies waarvan de waarde van het `Path`-attribuut het langst is, vooraan dienen te staan in de `Cookie`-header. In datzelfde document staat ook dat webservers niet moeten vertrouwen op de volgorde waarin de cookies staan in de ontvangen `Cookie`-header, wat het dan weer moeilijker maakt voor de aanvaller om te voorspellen welke cookie gebruikt zal worden. Hoe dan ook zal de aanvaller door middel van testen kunnen achterhalen welke cookie geprefereerd wordt.

Ook hier gaan we er weer vanuit dat de aanvaller een script kan injecteren op `benign-extended.com` of een van zijn subdomeinen, alsook dat hij de waarden van het `Path`- en `Domain`-attribuut kent. Zoals we in listing 5.3 tonen, heeft elke originele cookie geen specifiek `Path`-attribuut gekregen. De aanval voeren we uit door elke same-site cookie te *overschaduwen* door de nieuwe cookie een langer `Path`-attribuut te geven. In dat opzicht gaven we de nieuwe `__lax-sid` en `__strict-sid` respectievelijk de attributen `path=/only_lax` en `path=/only_strict`. Op deze manier werd de nieuwe cookie gebruikt door de webserver in plaats van de originele cookie, waardoor deze ook bij alle volgende cross-site requests werd meegestuurd. Het is belangrijk op te merken dat deze attack vector niet altijd kan toegepast worden. Zo zal de aanvaller in staat moeten zijn om zijn cross-site aanval uit te voeren op een webpagina met een `Path`-attribuut dat specifieker is dan dat van de originele cookie.

Nog een andere methode voor deze aanval is **cookie overflowing** [9]. De bedoeling is om zoveel mogelijk cookies aan te maken op de webbrowser van het slachtoffer vanaf het eigenlijke domein of een subdomein tot dat de limiet van die webbrowser overschreden is. Voor Chromium zijn dit bijvoorbeeld 180 cookies die per domein ingesteld kunnen worden. Hierbij worden ook alle cookies afkomstig van de subdomeinen opgeteld. Wanneer deze limiet bereikt is, zal Chromium plaats maken door de oudste cookies te verwijderen. Zo heeft de aanvaller plaats om gloednieuwe cookies in te stellen op basis van de cookiewaarden die hij voor de overflow heeft opgeslagen. Voor deze cookies kan hij het `SameSite`-attribuut dan achterwege laten.

In onze simulatie lieten we de aanvaller weer een script injecteren op `benign-extended.com` of een van zijn subdomeinen. Hij heeft hier echter niet de waarden van het `Domain`- en `Path`-attribuut nodig. Door op voorhand de originele cookiewaarden op te slaan in een variabele en daarna de overflow uit te voeren, slaagden we er in om vanaf een schone lei nieuwe cookies toe te voegen met dezelfde waarden, zonder `SameSite`-attribuut weliswaar.

5.2.5 Samenvatting

We hebben bestaande aanvallen gesimuleerd op `benign-extended.com`, een website die gebruik maakt van same-site cookies en de module die we beschreven hebben in hoofdstuk 4. Tabel 5.1 toont de resultaten van deze evaluatie. Per kolom voor de drie mappen duiden we aan of deze map kwetsbaar was voor de aanval in kwestie. De kolom van `/only_gen` gebruiken we ter referentie voor generische cookies. Naast de gesimuleerde aanvallen hebben we ook aangetoond dat same-site cookies onschadelijk gemaakt kunnen worden door aanvallen uit te voeren op sub- of superdomeinen van de doelwebsite en door middel van cookie tampering.

TABEL 5.1: Samenvatting van de evaluatie omtrent bestaande aanvallen.

	`benign-extended.com`		
	`only_gen`	`only_lax`	`only_strict`
CSRF			
Clickjacking			
- Hyperlinks	✓	✓	✗
- Embeddable content	✓	✗	✗
- Embedden in iframes	✓	✗	✗
Automatisch refereren			
- <img> en <iframe>	✓	✗	✗
- Redirects	✓	✓	✗
- <form> (GET)	✓	✓	✗
- <form> (POST)	✓	✗	✗
- XHR / Fetch API	✓	✗	✗
- Prerender	✓	✓	✗[1]
Login CSRF	✓	✓	✓
XSSI			
JSONP	✓	✗	✗
Dynamische JavaScript	✓	✗	✗
Identifier-based	✓	✗	✗
Timing attack			
Cross-site timing attack			
- Via JavaScript	✓	✗	✗
- Zonder JavaScript	✗	✗	✗
Browser-based timing attack			
- <audio>, <video>, <script>	✓	✗	✗
- Fetch API	✓	✗	✗
Size-exposing attack	✓	✗	✗

[1] De bug uit sectie 3.3.1 maakt het echter wel mogelijk om een `SameSite-strict` cookie te versturen in een cross-site request geïnitieerd door het prerender-element.

Hoofdstuk 6

Discussie

In sectie 3.3.1 voerden we een analyse uit om te achterhalen welke cross-site requests nu eigenlijk same-site cookies meesturen, afhankelijk van de instelling van die same-site cookies. We vonden hier een bug die ervoor zorgt dat `SameSite-strict` cookies meegestuurd worden in een cross-site request geïnitieerd door het prerender-element. Met deze bug kan een aanvaller gemakkelijk de bescherming van `SameSite-strict` cookies omzeilen. De Internet Draft specifieert de `SameSite-strict` cookie als de cookie die nooit met cross-site requests kan worden meegestuurd. Dit maakt deze same-site cookie aantrekkelijk om te gebruiken voor het authenticeren van impactvolle acties. Een schijnbaar drastische en veilige instelling kan op deze manier toch onschadelijk gemaakt worden. Met andere woorden haalt deze bug het hele concept van de `SameSite-strict` cookie onderuit. Daarom hopen we dat deze bug snel opgelost wordt, ook al zijn er nagenoeg nog geen websites die gebruik maken van same-site cookies. Het lijkt er anderzijds wel op dat de ondersteuning voor het prerender-element zal verdwijnen. Er is namelijk voorgesteld om deze functionaliteit te verwijderen [13]. Dit zou het probleem ook kunnen oplossen.

In de daaropvolgende secties van hoofdstuk 3, hebben we het over enkele inconsistenties die we gevonden hebben tijdens het onderzoek. Zulke inconsistenties maken de functionaliteit minder overzichtelijk en kunnen vaak over het hoofd gezien worden door ontwikkelaars, daar hier ook geen documentatie over te vinden is. Niet alleen zou een aanvaller deze inconsistenties naar zijn hand kunnen zetten, ze kunnen ook verwarrend zijn voor de gebruiker. Omwille van deze redenen vinden we het belangrijk dat deze inconsistenties in de toekomst opgelost worden of voldoende duidelijk gedocumenteerd worden. Zo kan de ontwikkelaar bijvoorbeeld toetsen of deze inconsistenties misbruikt kunnen worden in de context waarin hij de same-site cookie wilt gebruiken.

Het ontwikkelde server-side prototype evalueren we samen met de same-site cookie in hoofdstuk 5. Wat betreft performantie vereist het prototype een additionele CPU-rekentijd met een grootteorde van enkele microseconden. Het prototype draagt dus niet significant bij tot de vertraging van de responsetijd van de webserver, zelfs niet bij grote cookies. Het is wel zo dat dit prototype zorgt voor een toename van de grootte van de session cookie, bovenop het feit dat er drie cookies in plaats van

één cookie ingesteld worden. Hoewel deze toename in grootte significant kan zijn voor de gevallen waarin de originele cookie al redelijk groot is, is de kans klein dat maximale cookiegrootte overschreden wordt. In sectie 5.1.2 slaagden we er immers in om een originele cookie te gebruiken van 2900 bytes.

Uit de evaluatie op basis van bestaande cross-site aanvallen bleek dat same-site cookies een degelijke bescherming bieden tegen CSRF-aanvallen. Enkel Login CSRF wordt niet ingetoomd door de bescherming van same-site cookies. De oorzaak hiervan is dat same-site cookies wel nog ingesteld kunnen worden door responses op cross-site requests. Als oplossing stellen we voor dat het instellen van same-site cookies op deze manier ook gereguleerd wordt, conform met de regulering rond het meesturen van same-site cookies met cross-site requests. Anderzijds zijn er wel al beschermingsmaatregelen bekend die wel effectief zijn tegen Login CSRF. Via deze beschikbare beschermingsmaatregelen kan er dan ook opgetreden worden tegen deze specifieke aanval.

Wat betreft de gewone CSRF-aanvallen zijn het enkel attack vectors gebaseerd op GET-requests die een top-level navigatie veroorzaken of geïnitieerd zijn door het prerender-element die geauthenticeerd worden door `SameSite-lax` cookies. Zo een top-level navigatie maakt de aanval zeer zichtbaar en daarbij is het volgens de standaard eigenlijk ook niet de bedoeling om staatswijzigende acties te laten afhangen van GET-requests. De verdediging tegen CSRF-aanvallen op basis van same-site cookies zou dus veel effectiever zijn indien de website zich zou houden aan deze standaard, varianten zoals POST-requests krijgen namelijk geen same-site cookies mee. In dit geval zou geen enkele CSRF-aanval succesvol zijn geweest in deze evaluatie.

De XSSI-aanvallen, timing attacks en size-exposing attack werden allemaal met succes afgeweerd door het gebruik van same-site cookies. In tegenstelling tot CSRF, is hier minder ruimte om attack vectors te vinden die minder lijden onder de bescherming van same-site cookies. Voor een CSRF-aanval hoeft de request in kwestie gewoon maar ontvangen te worden door de webserver, terwijl deze aanvallen een bruikbare response dienen terug te krijgen. De *in-depth* bescherming die aangeboden wordt door same-site cookies is dus sterk genoeg om deze aanvallen onschadelijk te maken.

Hoewel er een goede weerstand wordt geboden tegen de meeste individuele aanvallen, zijn er wel manieren beschikbaar voor een aanvaller om same-site cookies te omzeilen. Zo worden requests tussen sub- en superdomeinen niet gezien als cross-site. Same-site cookies kunnen dus moeiteloos omzeild worden door de cross-site requests uit te voeren vanaf een sub- of superdomein van het doeldomein. Aangezien bepaalde firma's subdomeinen verhuren, zou de aanvaller op legale wijze de controle kunnen verkrijgen over een benodigd subdomein. Ook kan dit via een subdomein dat kwetsbaar is voor XSS-aanvallen, de attack surface is dus redelijk groot. Daarnaast zijn er ook verschillende manieren om cookies te manipuleren via deze attack surface zodat hun `SameSite`-attribuut wegvalt.

Hoofdstuk 7

Besluit

De same-site cookie wordt voorgesteld als nieuw *in-depth* verdedigingsmechanisme tegen cross-site aanvallen. Dit mechanisme wordt echter nog maar recent ondersteund door Chromium-afgeleiden Google Chrome en Opera. Daarnaast is er, naar onze kennis, nog geen wetenschappelijk onderzoek gepubliceerd hierover. In deze thesis onderzochten we de werking van dit mechanisme om zo over een beter overzicht te beschikken, omdat de Internet Draft hierover als enige bron informatie verschafte. Tevens ontwikkelden we een prototype met als doel de adoptie van same-site cookies te vergemakkelijken. Dit prototype evalueerden we in combinatie met de same-site cookie op basis van bestaande cross-site aanvallen.

Om een meer gedetailleerd beeld te krijgen over de werking van de same-site cookie, voerden we een analyse uit met behulp van de ondersteunende webbrowsers. Hieruit leerden we welke cross-site requests nu eigenlijk welke same-site cookies bevatten. Buiten één ontdekte bug, waren de resultaten in overeenkomst met de beschrijving in de Internet Draft. De aangetroffen bug kan gebruikt worden ter omzeiling van same-site cookies. We rapporteerden deze bug bij Chromium. Deze werd erkend maar is nog niet opgelost. Naast de analyse bevatte ons verkennend onderzoek ook nog de bespreking van gevonden inconsistenties. Deze inconsistenties zouden (in beperkte mate) uitgebuit kunnen worden in bepaalde situaties om same-site cookies te omzeilen. Daarnaast maken ze de functionaliteit van same-site cookies onoverzichtelijk voor zowel ontwikkelaar als gebruiker.

In deze thesis stellen we ook een prototype in de vorm van een servermodule voor dat tracht de adoptie van same-site cookies te vergemakkelijken. We slaagden erin de implementatie van de website volledig af te schermen van de implementatie van same-site cookies. Met de lage CPU-kost verhoogt de module de last van de server niet significant. Daarbij is dit prototype breed toepasbaar omdat het als servermodule voor de meeste webservers geïmplementeerd kan worden. Daarnaast is het ook geschikt als oplossing voor applicaties waar het moeilijk is om broncode te wijzigen zoals legacy-applicaties en contentmanagementsystemen.

We evalueerden de same-site cookie in een testomgeving die gebruik maakte van ons prototype. Uit de resultaten volgde dat de same-site cookie een antwoord biedt op de vraag naar verdediging tegen de meeste cross-site aanvallen. Zowel

XSSI, cross-site timing attacks als size-exposing attacks werden volledig onschadelijk gemaakt door gebruik van same-site cookies. Ook verschillende attack vectors voor CSRF konden niet meer gebruikt worden voor een succesvolle aanval. Daarbuiten was enkel de `SameSite-lax` cookie vatbaar voor sommige CSRF-aanvallen. Dit was het geval voor redelijk zichtbare aanvallen, alsook voor het gebruik van prerender-functionaliteit eigen aan Chromium. De ontdekte bug zorgt er evenwel voor dat ook `SameSite-strict` cookies vatbaar zijn in dit laatste geval. Het gaat hier echter allemaal om GET-requests, waardoor het redelijk eenvoudig is voor een website om dit probleem op te lossen door staatswijzigende acties enkel toe te laten als gevolg van POST-requests. We toonden in diezelfde evaluatie aan dat Login CSRF niet afgeweerd kan worden door bescherming op basis van same-site cookies. Er zijn echter al maatregelen bekend die wel voldoende bescherming bieden. Anderzijds beschrijven we ook een wijziging in de werking van same-site cookies die dit probleem zou kunnen verhelpen.

Ten slotte identificeerden we ook enkele manieren waarop de bescherming van same-site cookies omzeild kan worden. Zo tonen we aan dat de attack surface hiervoor zo groot is als de verzameling van sub- en superdomeinen van het doeldomein. De aanvaller zal namelijk de same-site cookie regulering vlotjes kunnen omzeilen als hij een script kan injecteren op zo een gerelateerd domein. Daarnaast tonen we ook aan dat same-site cookies gemanipuleerd kunnen worden zodat ze hun `SameSite`-attribuut verliezen. Dit is ook mogelijk door een script te injecteren via de voorgenoemde attack surface.

Bijlagen

Bijlage A

Volledige verzameling van de analyseresultaten uit sectie 3.3.1

TABEL A.1: Cookies meegestuurd met cross-site requests (volledig).

Webbrowser-engine	Blink			Gecko		
Webbrowser(s)	Google Chrome Opera			Mozilla Firefox		
Cookie	gen	lax	strict	gen	lax	strict
a-href*	✓	✓	✗	✓	✓	✓
a-ping*[1]	✓	✗	✗	✓	✓	✓
audio-source-src	✓	✗	✗	✓	✓	✓
body-background	✓	✗	✗	✓	✓	✓
button-formaction*	✓	✓	✗	✓	✓	✓
css-after-content-2	✓	✗	✗	✓	✓	✓
css-background-image	✓	✗	✗	✓	✓	✓
css-before-content-2	✓	✗	✗	✓	✓	✓
css-border-image-source	✓	✗	✗	✓	✓	✓
css-cursor	✓	✗	✗	✓	✓	✓
css-escape-url-1	✓	✗	✗	✓	✓	✓
css-escape-url-2	✓	✗	✗	✓	✓	✓
css-font-face-src	✗	✗	✗	✗	✗	✗
css-font-face-src-ttf	✗	✗	✗	✗	✗	✗
css-font-face-src-woff	✗	✗	✗	✗	✗	✗
css-import-string	✓	✗	✗	✓	✓	✓
css-import-url	✓	✗	✗	✓	✓	✓
css-list-style-image	✓	✗	✗	-	-	-
css-shape-outside	✗	✗	✗	-	-	-
css-variables	✓	✗	✗	✓	✓	✓

[1] De URL werd gespecifieerd door het ping-attribuut

Webbrowser-engine	Blink			Gecko		
Webbrowser(s)	Google Chrome Opera			Mozilla Firefox		
Cookie	gen	lax	strict	gen	lax	strict
embed-src	✓	✗	✗	✓	✓	✓
iframe-javascript-src	✓	✗	✗	✓	✓	✓
iframe-javascript-src-2	✓	✗	✗	✓	✓	✓
iframe-src	✓	✗	✗	✓	✓	✓
iframe-srcdoc-img-src	✓	✗	✗	✓	✓	✓
image-src	✓	✗	✗	✓	✓	✓
img-src	✓	✗	✗	✓	✓	✓
img-srcset	✓	✗	✗	✓	✓	✓
inline-css-background-image	✓	✗	✗	✓	✓	✓
inline-css-border-image-source	✓	✗	✗	✓	✓	✓
inline-css-cursor	✓	✗	✗	✓	✓	✓
inline-css-list-style-image	✓	✗	✗	-	-	-
inline-css-shape-outside	✗	✗	✗	-	-	-
input-src	✓	✗	✗	✓	✓	✓
isindex-src	-	-	-	✓	✓	✓
link-alternate-stylesheet	✓	✗	✗	-	-	-
link-import	✗	✗	✗	-	-	-
link-prefetch	✓	✗	✗	-	-	-
link-preload	✓	✗	✗	-	-	-
link-preload-as-font	✓	✗	✗	-	-	-
link-preload-as-image	✓	✗	✗	-	-	-
link-preload-as-script	✓	✗	✗	-	-	-
link-preload-as-style	✓	✗	✗	-	-	-
link-prerender	✓	✓	✓	-	-	-
link-shortcut-icon	-	-	-	✓	✓	✓
link-stylesheet	✓	✗	✗	✓	✓	✓
meta-refresh	✓	✓	✗	✓	✓	✓
object-data	✓	✗	✗	✓	✓	✓
object-data-x-scriptlet	✓	✗	✗	✓	✓	✓
picture-img-srcset	✓	✗	✗	✓	✓	✓
script-src	✓	✗	✗	✓	✓	✓
svg-a-text*	✓	✓	✗	✓	✓	✓
svg-cursor	✓	✗	✗	✓	✓	✓
svg-feimage	✓	✗	✗	✓	✓	✓
svg-image-href	✓	✗	✗	✓	✓	✓
svg-image-set	✓	✗	✗	✓	✓	✓
svg-image-xlink-href	✓	✗	✗	✓	✓	✓
svg-script-href	✓	✗	✗	✓	✓	✓
svg-script-xlink-href	✓	✗	✗	✓	✓	✓

Webbrowser-engine	Blink			Gecko		
Webbrowser(s)	Google Chrome Opera			Mozilla Firefox		
Cookie	gen	lax	strict	gen	lax	strict
table-background	✓	✗	✗	✓	✓	✓
td-background	✓	✗	✗	✓	✓	✓
track-src	-	-	-	✓	✓	✓
video-poster	✓	✗	✗	✓	✓	✓
video-poster-2	✓	✗	✗	✓	✓	✓
video-source-src	✓	✗	✗	✓	✓	✓
video-src	✓	✗	✗	✓	✓	✓

Bijlage B

Populariserend artikel

COOKIES
WITH AN AFTERTASTE

By Gertjan Franken

Cookies have become a significant part of the Internet that we know of today. It all began when the cookie was first introduced in 1994 by the now discontinued web browser Netscape Navigator. The purpose of the cookie was to store small bits of data on the user's computer. This way, state information could be stored client-side instead of server-side. This new invention was a success and other browsers started supporting cookies shortly after. From then on, cookie functionality kept evolving and new variants were introduced such as the Flash cookie.
Unfortunately, the cookie is not as harmless as it seemed. Even back in 1996, cookies got a lot of media attention when it became known to the public that their browser actually stores data on their computer. Cookies can be used in a lot of ways, of which some are not in the best interest of the user. In this article I will discuss some of the most dangerous exploits that can be used on cookies. I will end with an introduction to the same-site cookie, a promising new security concept.

TRACKING

One of the reasons that cookies became widely known to the public in 1996 was an article of Tim Jackson in the Financial Times[1]. In his article "*This bug in your PC is a smart cookie"*, he criticized the potential privacy threat that cookies impose on consumers. He even called cookies the silver bullet for companies to target consumers individually. Companies would now know exactly how many times a user has visited its website and what pages he preferred, by simply using cookies. Although this was already somewhat possible by other means such as IP address checking, cookies provided a lot more precision.

Third-party cookie

But it gets worse and the one to blame it the *third-party cookie*. This kind of cookie was already being used not long after the birth of the original cookie. When a site embeds content from another site, your browser will fetch this content by sending a request to that other site. This

[1] Jackson, T. "This bug in your PC is a smart cookie", Financial Times, February 12th 1996, p. 2.

other site can decide for your browser to store a cookie, a third-party cookie. Now, when you visit another site that also happens to include content from that same site, another cookie can be stored. Subsequently this site has stored two distinct cookies that will be sent with every request to this site. As a result, it becomes easy to track users across multiple web domains. Imagine an advertising company of which its advertisements are embedded on a lot of different websites. Third-party cookies would provide them with a lot of personal information. Most modern browsers let users block third-party cookies, but unfortunately this is not done by default. One can check which cookies (first-party and third-party) are stored for a particular website at cookie-checker.com.

Zombie cookies

A sensible solution would be to just delete those harmful cookies. This would be easy if it weren't for the *zombie cookie*. First, let me give you a little background. There is a whole range of different storing mechanisms that can be used to store cookie data, for example local shared objects (Flash cookies) and HTML5 Web storage. In essence, cookies can be stored in a lot of different locations on the client's PC and some of these are even difficult to access for the user. In 2010, Samy Kamkar created Evercookie[2]. This is a JavaScript-based application that makes cookies extremely persistent by using different storing mechanisms to save a cookie. Briefly explained, a website using Evercookie will request your browser to store a lot of different cookies, each one using a different storing mechanism. But all these cookies are actually duplicates of one original cookie. When the user deletes one of these duplicates, this application will find one of its duplicates stored at another place and restore the deleted cookie. The cookie gets, as it were, resurrected. Hence, it is named a zombie cookie. Not only commercial websites and advertising companies see benefit in zombie cookies. According to documents leaked by Edward Snowden in 2013, the NSA cited Evercookie as a means to track Tor users.[3]

Cookie syncing

Zombie cookies aren't the only way to make cookies more persistent. *Cookie syncing* is also up for that job. An important property of cookies is that they are domain-specific. That is, a cookie belongs to one and only one domain. So in retrospect, a cookie can not belong to foo.com and bar.com at the same time. Yet cookie syncing can be used to overcome this problem such that multiple websites, belonging to different companies, can share their cookies. Advertising companies for instance can make deals to manage shared databases filled with cookie data.

[2] http://samy.pl/evercookie/
[3] https://edwardsnowden.com/docs/docs/tor-stinks-presentation.pdf

Among those advertising companies, user data can be merged by mapping corresponding user IDs together. As a result, users can be tracked on all participating domains.

HACKING

Usability is notably enhanced by the use of cookies. A cookie can be used to identify the session of a logged in user. The user is not required to authenticate himself for every action he performs on a certain website. The most common way to implement this is to set a cookie with a random and unique session ID when the user logs in. This session ID is also stored server-side and is linked to the logged in account. As a result, every request carrying this cookie will be seen as a request from this user and thus the user does not have to manually authenticate himself each time. This, however, poses some serious risks.

Cross-site request forgery

According to OWASP, *cross-site request forgery* (CSRF) is still one of the most prevalent web attacks.[4] The adversary designs a website in the hopes an unaware victim visits it. This website will embed content of a benign site in the form of, for example, an <img>-tag. The victim's browser will automatically fetch the image by using the provided source. The particular adversary is smart and puts `<img src="www.bank.com?transferAmount=5000&recipient=baddy">` in his HTML page. Let's say that the bank's website is not really secure and accepts GET parameters. Visiting this image source will transfer 5000 dollars to the account of baddy. Of course, only an authenticated user can transfer money from his account. This is where the cookie comes into play. If the victim is logged into his bank account, the associated cookie will be sent along with the request for the image. In turn, the bank will see this as an innocent money transfer request. It's not this easy in reality, but it works in the same way.

Luckily, defense mechanisms have been developed to provide protection against CSRF attacks. One of those is the synchronizer token pattern. A secret code is hidden in the HTML page of the benign website. This secret code, called the CSRF token, is a large and random value that is unique for every user session. It's more or less impossible for the adversary to guess this secret token. Each state changing request from the user is expected to carry this secret token. This is only possible when the user performs the request on the HTML page of the benign website, because this is where the CSRF token resides. Thus when the user's browser makes this request from the adversary's website, it will not carry the necessary CSRF token and therefore the benign website will reject the user's request.

[4] https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet

Timing attacks

Imagine you open your browser and surf to facebook.com. In case you are logged in and the necessary cookies are sent with the request, your news feed will be shown. In the other case (e.g. when you have no account or when cookies are disabled), the login page will be shown. There is obviously a huge difference between your news feed and the generic login page of Facebook. Most importantly, the news feed contains a lot more content. Now, say that an adversary has the opportunity to measure the time necessary to request, receive and process your requested webpage. This is possible when, for example, you visit the adversary's website which embeds content of facebook.com. It is not that difficult to determine whether you are logged in or not. Of course timing this is subject to noise such as fluctuations in connection speed. Statistics can pose a remedy for this problem, but there's also another variant of timing attacks that circumvent this, namely browser-based timing attacks. This variant concentrates on side-channel leaks of browsers and are not subject to unstable network connections. Timing starts when the resource has been fully downloaded and ends when it has been processed. It has been demonstrated that the adversary is able to further specify the gender, age and location by using targeted Facebook ads.[5]

Cross-site script inclusion

The last attack I want to discuss in this document is *cross-site script inclusion* (XSSI). This attack exploits dynamically generated JavaScript scripts. These scripts are used to personalize a user's experience on a website. The script is automatically generated when given certain properties of a user account. These properties can be the username, e-mail address or the preferred language of a user. An adversary is able to import this script on his own web page using the <script>-tag. When a user with a cookie visits this malicious site, his browser will fetch the dynamically generated script. The script gets generated based on the cookie. The cookie authenticates the user for whom the script is generated. The source code of this script is not accessible for the adversary, but unfortunately it gets executed within the environment of his web page. For instance, he can access the global variables and overwrite global functions to which sensitive information is passed. Ultimately, he is able to save the gathered information for himself.

[5] Van Goethem, T., Joosen, W., & Nikiforakis, N. (2015). The Clock is Still Ticking. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15. doi:10.1145/2810103.2813632

Same-site cookies

The problems mentioned above can definitely count on some resistance from already-established security mechanisms. Nonetheless, there's always room for new security concepts. One of these promising new concepts is the same-site cookie. All of the exploits mentioned above are based on cross-site communication. To be more precise, they are based on the cookie connected to this cross-site communication. This is where the same-site cookie concept focuses on.

How it works

The same-site cookie was introduced by Google and eventually implemented in 2016.[6] Same-site cookies have an extra attribute called *SameSite*. This attribute can have two values: *strict* and *lax*. In the case of strict, the cookie will never be sent along with a cross-site request. In other words, the cookie will only be sent along with requests that originate from the same domain or when you type it manually in the address bar of your browser. So for example when you have an account on twitter.com and this cookie is set to strict, the cookie will not be sent along with the request when you click on a link to twitter.com on another website. Therefore you will not appear to be logged in. However, when you refresh the page you will be logged in. The attribute value lax is more lenient. When your same-site cookie is set to lax, it will still be sent along with cross-site requests that meet the requirements. Basically, it is only sent along with a GET request that is top-level (i.e. the URL in your browser's address bar changes due to this navigation).

The adoption

As of yet, same-site cookies are only supported by Google Chrome and Opera.[7] All other browsers will just ignore the SameSite attribute. Clearly, same-site cookies will break some functionality when a website implements it. Because of its novelty, virtually no website has done this yet. This makes it difficult to measure the difficulties of implementing same-site cookies. The combination of future research and transitioning websites will surely shed a broader light on this.

[6] https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site-00
[7] https://www.chromestatus.com/feature/4672634709082112

Bijlage C

Wetenschappelijk artikel

Obstacles and Vulnerabilities Concerning the Adoption of Same-site Cookies

Gertjan Franken
Katholieke Universiteit Leuven
3001 Leuven, België
gertjan.franken@student.kuleuven.be

Tom Van Goethem
Katholieke Universiteit Leuven
3001 Leuven, België
tom.vangoethem@cs.kuleuven.be

Wouter Joosen
Katholieke Universiteit Leuven
3001 Leuven, België
wouter.joosen@cs.kuleuven.be

***Abstract*—Cross-site attacks still pose a serious threat against users of web applications, despite the numerous server-side and client-side solutions. A new in-depth defense mechanism has been introduced against these kinds of attacks, called the same-site cookie. Thus far, Google Chrome and Opera are the only popular web browsers to support same-site cookies. This does not create the necessary incentive for websites to make use of this new functionality.**

In this paper, we conduct an exploratory research on the same-site cookie and its behavior. This way we try to clarify the less detailed parts of the same-site cookie Internet Draft. We also developed a prototype to assist website administrators in the integration of same-site cookies for session cookies, with the intention to ease adoption. Finally, we will evaluate the same-site cookie in terms security, based on known attacks.

I. Introduction

The Internet has evolved into a strongly intertwined collection of websites that reuse all sorts of resources from one another. Resources such as images, advertisements and scripts are being shared effortlessly across different domains. One has to solely reference to a specific resource by means of an HTML tag and the web browser will fetch it through a request. If this resource resides on another domain, this request is called a cross-site request. Although cross-site requests enable quite some useful functionality, they are also used to construct attacks. Often these cross-site requests contain a form of implicit authentication, like a cookie. A malicious website could ensure for such an authenticated cross-site request to be sent automatically by the web browser that visits the website.

The most infamous attack that applies this principle is *Cross-Site Request Forgery* (CSRF), placed eighth in the *OWASP Top Ten Project* of 2013 [1]. Through his own website, the attacker will try to force a user's web browser to send a state-changing request to a web server this user is currently authenticated for. Because the web server assumes the victim himself intended to send this request, it will execute the requested state-changing action. Vulnerabilities have already been found in the past for popular websites like Youtube, The New York Times, ING Direct and MetaFilter by Zeller and Felten [2]. Multiple server-side solutions against this attack already exist such as the synchronizer token pattern and `Referer`/`Origin` header checking, but also client-side browser extensions like CsFire focus on the mitigation of this attack [3].

Login CSRF is a variant of CSRF, introduced by Barth et al. [4]. In this case, the attacker forces the victim's web browser to log in to the attacker's account of a web application. This can be achieved by initiating a CSRF attack which logs the user in to the account. The attacker's intention is for the user to remain logged in implicitly (e.g. by the means of a session cookie) without noticing being logged in to a malicious account. The user will use the web application just like he would any other day and afterwards the attacker will be able to effortlessly collect the data affected by the user's actions. Search terms and personal information could have been submitted by the user, which are now accessible by the attacker. In their paper, Barth et al. also examine existing defenses against CSRF and login CSRF attacks. They recommend strict `Referer` header validation for defense against login CSRF, while synchronizer tokens also mitigate the attack if applied before login.

Cross-site requests and implicit authentication can also be employed in an attack to steal sensitive information. *Cross-Site Script Inclusion* (XSSI) is such an attack that tries to steal user specific information through JSONP. JSONP enables a way to circumvent the *Same-Origin Policy* in order to access data in JSON format that originates from another domain. An authenticated request can then be used to access information about the user in question. A script on the attacker's website can facilitate the attack by forcing the web browser to send an authenticated cross-site request. The information in the response can then be leaked by that same script. Lekies et al. showed that dynamically generated JavaScript can also be an effective target for this attack [5]. In that same paper, Lekies et al. propose some sensible rules to protect a web application against XSSI. In short, these rules dictate that a script should be static and separated from any sensitive and dynamic data.

Cross-site requests also enable side-channel leaks to be exploited. Bortz and Boneh showed that cross-site timing attacks are an effective means to construct personal information about users [6]. On the basis of time measurements and applied statistics, they could deduce how much items were present in a user's online shopping cart. This attack, influenced by external factors such as network speed fluctuations, is more difficult to execute in practice. Bortz and Boneh also implemented an Apache module which tries to regulate the timings of responses. This way, it becomes more difficult for the attacker

to see a usable distinction between multiple responses. They also mention that merely adding random delays does not thwart these timing attacks.

Van Goethem et al. have found a variant of the cross-site timing attack, the browser-based timing attack, which is not influenced by network speed fluctuations [7]. The time measurements used for this variant do not take into account the time necessary to download a cross-origin resource, only the processing time of the web browser. This makes the attack more reliable and also bypasses existing defense mechanisms against cross-site timing attacks. Van Goethem et al. also mention that checking the `Referer` and `Origin` header in combination with a placeholder web page could be an effective defense against this attack.

In this paper, we examine a new defense mechanism against the mentioned attacks. This defense mechanism is the same-site cookie [8], supported by Google Chrome and Opera since 2016. Same-site cookies have an extra attribute which specifies which cross-site requests are allowed to carry this cookie. This policy is enforced by the web browser and set by the web server. Unfortunately, the adoption of the same-site cookie by websites is slow, as other popular web browsers such as Mozilla Firefox, Microsoft Edge and Safari do not offer support. Besides, the adoption requires a re-evaluation of the website's present cookie strategy to fit in this new defense functionality. We propose a prototype in the form of an Apache module that eases the adoption for websites. This module, in combination with the same-site cookie, is then evaluated in terms of security against the discussed cross-site attacks.

II. Background

The same-site cookie has an extra attribute in comparison to a generic cookie. This is the `SameSite` attribute, which accepts two distinct values: *strict* and *lax*. In case of the value of strict, the cookie will not be included in any cross-site request by a supporting browser. We call this a `SameSite-strict` cookie. In case of the value of lax, the cookie will only be sent along with a cross-site request if this request meets certain criteria. The rule of thumb here is that the request has to be a GET request that causes a top-level navigation. We call this a `SameSite-lax` cookie. For example, a click on a hyperlink will cause a cross-site request carrying `SameSite-lax` cookies. However, an <img> tag will cause a cross-site request of which `SameSite-lax` cookies are excluded because no top-level navigation will occur. The `SameSite-strict` cookies will be excluded in both these examples.

It is also important to notice what according to the Internet Draft is exactly considered a same-site and cross-site request. It states that a same-site request requires the initiating website of this request to have the same *registrable domain* as the target URI. The *registrable domain* is in this case the host's public suffix (e.g. .com, .co.uk, .be) plus the label to its left. So the *registrable domain* of `http://www.sub.example.com/` would be `example.com`. Notice that there is no distinction between differing subdomains.

III. Exploratory research

To obtain a more thorough understanding of same-site cookies, we performed some exploratory research in order to discover the exact behavior of this new defense mechanism.

A. Analysis of cross-site requests

There are many ways for a HTML page to leak information through cross-site requests. The GitHub repository HTTPLeaks[1] tries to collect all those HTML elements that initiate a HTTP request into one single HTML page. We used this page to investigate which same-site cookies are actually sent along with which cross-site requests. These tests were run with the two same-site cookie supporting web browsers, Google Chrome and Opera, both derivatives of the open-source project Chromium. Both these web browsers gave identical results.

The most unexpected discovery was that both the `SameSite-lax` and `SameSite-strict` cookies were included in cross-site requests initiated by `prerender` [9] functionality. By using `prerender`, a website can force a supporting web browser to preemptively render an indicated web page in the background. This will enhance the loading time when this web page is later requested through user interaction. The same-site cookie Internet Draft states that `SameSite-lax` cookies are included in a request initiated by `prerender` as an exception. However, this is not stated for `SameSite-strict` cookies, while our experiments indicate that this cookie was also included in these cross-site requests. We submitted this discovery as a bug report for Chromium. The bug has been acknowledged and the issue will be fixed.

Other results were in accordance with the description of the Internet Draft. Our analysis confirms that HTML elements such as the <img>, <iframe> and <script> tag cannot initiate cross-site requests that include any same-site cookie. Hyperlinks and meta-fresh, however, will initiate cross-site requests carrying the `SameSite-lax` cookie.

B. Subdomains

As was already mentioned earlier, the same-site cookie Internet Draft makes no distinction between subdomains concerning same-site requests. So in essence a request originating from one subdomain that is destined for another subdomain with the same parent domain, is considered a same-site request. Also requests between parent domain and subdomain are considered same-site requests. This is not in accordance with how `Domain` attributes are matched, stipulated by RFC 6265[10].

In our analysis, we set up a domain with two subdomains as is depicted in figure 1. All possible requests between these three domains depicted by the green arrows, were treated as same-site requests. As a consequence, they all carried every same-site cookie, no matter how the request was initiated.

[1] https://github.com/cure53/HTTPLeaks

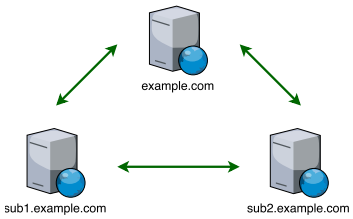


Fig. 1. Every request between the three domains is seen as a same-site request.

C. Server-side redirects vs. client-side redirects

Websites can refer to another web page by using a redirect. The web browser will then automatically fetch and render the referred web page. A server-side redirect can be invoked by a response carrying a `30x` status code (RFC 7231 [11]). A client-side redirect can be invoked by using the HTML <meta>-tag or JavaScript.

We found that a same-site cookie behaves differently depending on the kind of redirect, as a response to a cross-site request. When the web browser encounters a server-side redirect as a response to an initial cross-site request, the context of the redirected request will be the same as the context of the initial request. As a result, the same same-site cookies will be included in the redirected request as in the initial request. This is illustrated in figure 2. The generic cookie and the `SameSite-lax` cookie included in the initial request are also included in the redirected request to `example.com`. Thus the redirected request is considered a cross-site request, just like the initial request.

Another behavior is achieved when a client-side redirect is used. The redirected request will be sent in the context of the web page on which the redirect reference is placed. This is illustrated in figure 3. Because the redirected request is initiated in the context of `example.com`, it is considered a same-site request. Therefore, the `SameSite-strict` cookie is included in the redirected request, in addition to the other two cookies.

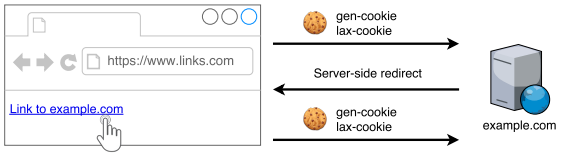


Fig. 2. An example in which a server-side redirect causes the redirected request to stay cross-site.

It is important to note that this is only applicable if a top-level navigation is made. This scenario cannot be reproduced by using for example <img> or <iframe> tags for the initial request.

D. Other findings

Another little quirk we found is the inconsistency between ways to open a link. When presented a hyperlink, the user can choose to open this link in the current tab, in a new tab or even in a new window. These options can be accessed by right-clicking the link or using shortcuts. As we've already mentioned, a cross-site request initiated by a hyperlink will carry `SameSite-lax` cookies, but not `SameSite-strict` cookies. This is exactly what happens when the user opens the link in the current tab. However, when the user chooses to open the link in a new tab or in a new window, the initiated request will also carry the `SameSite-strict` cookies. We suspect the reason for this may be that the request is made from a newly created context, which makes the request same-site.

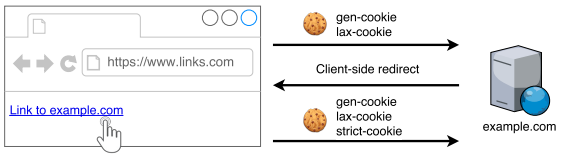


Fig. 3. An example in which a client-side request causes the redirected request to be same-site.

IV. Prototype for easing adoption

In order to make the adoption easier, we developed a prototype in the form of an Apache module. This module focuses on integrating the same-site cookie functionality for a session cookie. This way, we try to offer the website administrator a more fine-grained control over the session cookie, dependent on the impact level of certain endpoints of his website.

The concept of this module is based on the splitting of the session cookie into three separate session cookies. One session cookie will be generic, one will be `SameSite-lax` and one will be `SameSite-strict`. Each of these cookies will carry the encrypted value of the original session cookie, each one encrypted with a different key. For every response, the module will split the original session cookie in the `Set-Cookie` header into the three converted session cookies by using encryption. The other way around, the module will deconvert the appropriate converted session cookie in the `Cookie` header for every request. This process is depicted in figure 4. Here, the session cookie `sid` has been chosen to be the assigned original session cookie. We made use of *cookie prefixes* [12] to name the converted cookies.

By using the Apache configuration files, the web administrator is able to define which session cookie will be converted and deconverted by the module. The deconversion is context-dependent, so the administrator will also have to define which one of the three converted cookies in the `Cookie` header will suffice for a certain endpoint on his website. He could decide for his website `example.com` that functionality accessible at `example.com/about` only requires the generic converted cookie, while functionality at `example.com/settings` requires the `SameSite-strict` cookie to perform a state-altering action. In a particular scenario, `example.com/settings`

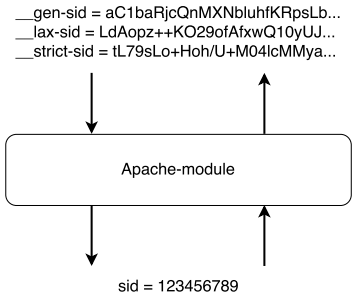


Fig. 4. An example of the concept of the developed Apache module.

could receive a request lacking the `SameSite-strict` cookie because it was excluded by the web browser. Then no cookie will be deconverted as though the web server did not receive the session cookie at all. This functionality makes it possible for a website to accept one of these three session cookies based on the impact an attack would have on a certain endpoint.

We tested the performance of this module in order to estimate the potential load on servers to run this module. This test was based on deconversion of converted cookies in the `Cookie` header, because deconversions will occur the most. The necessary CPU time was measured for 120 requests of which each one required a cookie to be deconverted. Figure 5 depicts a the boxplots for three separate tests for a cookie size of 900, 1900 and 2900 bytes. The mean values for these tests are 140,6 µs, 171,2 µs and 212,5 µs respectively.

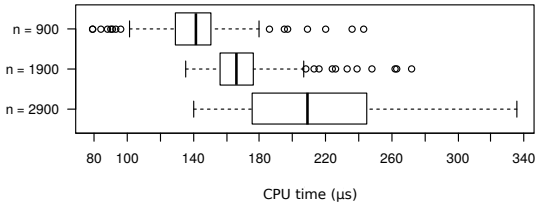


Fig. 5. The boxplots showing the results of the performance tests for cookie sizes 900, 1900 and 2900 bytes.

V. Evaluation

We already mentioned the three major cross-site attacks against which the same-site cookie is intended to offer an extra layer of defense. In this section, we evaluate the effectiveness of the same-site cookie against these attacks.

A. Cross-Site Request Forgery

CSRF attacks can be initiated by a lot of means. Based on existing attack vectors and functionality specific to the same-site cookie, we evaluate the level of defense provided by the same-site cookie.

1) Clickjacking: Clickjackers try to trick the user into clicking on a specific button which causes the web browser to execute an unintended action. Different techniques that can facilitate such an attack have already been researched [13]. These can be used to initiate a CSRF attack (e.g. clicking on a malicious hyperlink). As we've already mentioned, `SameSite-lax` cookies will be sent along with cross-site GET requests that cause a top-level navigation. Clicking on a hyperlink will cause the web browser to send such a request. Thus `SameSite-lax` cookies can be exploited in this way. The biggest drawback of this attack is its visibility, the web browser will load the referenced web page after all. Huang et al. mention existing defensive measures specifically for clickjacking such as UI randomization and extra user confirmation [13].

2) Automatic references: CSRF attacks are often executed by using HTML elements that refer to a resource. <img> and <iframe> are elements often used for this purpose. The web browser will automatically send a request to fetch the made-up resource, using the included malicious URL. Fortunately, same-site cookies are not included in these cross-site requests, which will deviate the attack.

Automatic redirects can also be used to carry out a CSRF attack. Because redirects cause a top-level navigation, `SameSite-lax` cookies will be included in this cross-site request. This poses an alternative for the clickjacking attack vector. An automatic redirect can also be simulated by a <form> through GET and submitting it automatically by using JavaScript. Just as with clickjacking, the biggest drawback here is the visibility of the attack.

Our analysis in section III-A uncovered a bug which causes the web browser to include all same-site cookies for a cross-site request initiated by `prerender`. This undermines the entire concept of the `SameSite-strict` cookie, being that it will never be included in a cross-site request. Both Google Chrome and Opera which support same-site cookies, also support `prerender`. This weakens the extra layer of defense that the same-site cookie was intended for. Considering that this bug will be fixed, `SameSite-lax` cookies are still intended to be included in a cross-site request initiated by `prerender`. This forms an attractive alternative for the attack vectors mentioned in this section, because it still includes `SameSite-lax` cookies though an automatic attack that is invisible for the user.

3) Login CSRF: Same-site cookies on its own do not realise an appropriate defense against login CSRF. Consider the case in which a benign website uses a `SameSite-strict` session cookie, the most defensive setting. It is still possible for the attacker to force the user to log in to his account, while the user will remain authenticated through that session cookie. The same-site cookie will be set just like any other cookie. Like we already mentioned, the appropriate defense according to Barth et al. would be strict `Referer` header checking [4].

4) Through subdomains: Like we explained in section III-B, requests between subdomains and parent domains are seen as same-site requests by the same-site cookie Internet Draft. This opens up some opportunities for the attacker to organise a CSRF attack that bypasses the same-site cookie defense. Instead of trying to inject malicious code into the target domain in order to facilitate a CSRF attack, he now has the possibility to inject it into a subdomain or parent domain of the target domain. In other words, the attack surface to bypass same-site cookies does not only contain the target domain, but also its affiliated domains. Besides injecting code into a certain web page, the attacker could also obtain control of a subdomain through legal action if available. *Content Delivery Network* (CDN) providers often offer subdomains of a shared domain to their customers. This was also mentioned by Zheng et al. concerning the dangers of cookie injection attacks [14]. An attacker could easily bypass same-site cookie defenses if able to pose as a customer for a certain subdomain.

B. Cross-Site Script Inclusion

By using XSSI, the attacker attempts to steal user specific information residing on a web server. This attack can be executed by using JSONP vulnerabilities, but also by exploiting dynamically generated JavaScript as discussed by Lekies et al. [5]. These attacks are carried out through the use of scripts on the malicious web page. Scripts used by the target website are imported using the <script> tags. As we've mentioned in section III-A, no same-site cookies are included in cross-site requests that are initiated in order to import a script. This is also confirmed by the attack that we simulated on a domain that implemented same-site cookies. The request remains unauthenticated without the session cookie, thus same-site cookies appear to be a robust defense mechanism against XSSI-attacks.

However, the attacker could try to ensure that these scripts are not meant to be imported through cross-site requests. If the scripts could be imported with same-site requests, the attack would bypass same-site cookies. We could use the same principle as we've used in section V-A4, namely exploiting subdomains. If the attacker could obtain control of a page on a related domain, or the entire subdomain, same-site cookies would be bypassed.

C. Cross-Site Timing Attacks

Cross-site timing attacks are used to construct and obtain information about a victim (e.g. whether he has an account on a specific website or how many items are in his online shopping cart). The attacker forces the victim's web browser to send requests to a benign website in order to measure how much time was needed to reply. Bortz and Boneh describe their primary technique as using invisible images through the <img> tag and JavaScript to perform multiple timings [6]. Same-site cookies won't be included in cross-site requests initiated by an <img> tag, as we've discussed in section III-A. The cross-site requests will thus not be authenticated and the server will not reply with user-specific content. We can say that the use of same-site cookies successfully avert this attack, because the web server does not leak information when responding to unauthenticated requests.

Browser-based timing attacks are a recent variant of the cross-site timing attack. Instead of exploiting the time to download a resource, the time needed for the web browser to parse a resource is exploited. Research by Van Goethem et al. shows that a wide variety of web browser timing side-channels can be used for this [7]. The paper mentions attacks through <audio>, <video> and <script> tags, together with enhancements by ApplicationCache and Service Workers. Our analysis explained already that <script> tags do not initiate cross-site request that include any same-site cookie. We can add to this that the same applies to <audio> and <video> tags. Other than that, we also tested the fetch API used by Server Workers. Cross-site requests initiated by fetch() never included any same-site cookies, so same-site cookies can be a solution against browser-based timing attacks. We did not test ApplicationCache since it has been deprecated in favor of Service Workers.

Also here, we must mention that same-site cookies can be bypassed for requests between subdomains and parent domains.

VI. Discussion

In this paper, we proposed a prototype that could help website administrators to integrate same-site cookies in their session cookie strategy. As this prototype does not require any session-specific data to be saved server-side and the processing time of each request is only increased in the order of microseconds, we believe this prototype can be a valuable extension to a web server for using same-site cookies. We believe this is the first prototype developed with the intent to ease the adoption of same-site cookies.

As for the evaluation of the same-site cookie, it showed that the same-site cookie provides an appropriate level of defense against CSRF, XSSI and timing attacks. Not taking into account the `prerender` bug we found, the `SameSite-strict` cookie will never be included in a cross-site request. This is a powerful tool against the discussed cross-site attacks. Of course not just any cookie can just be made `SameSite-strict` because it will break existing web application functionality. Besides this, the usability will also decrease as users will not be automatically logged in anymore for a lot of use cases. Luckily, our prototype makes it easy to accept same-site cookies according to context-specific settings at endpoints. Although the `SameSite-lax` cookie can be abused if intended to authenticate state-altering actions through GET requests, it's still a solid defense through POST requests.

Unfortunately, login CSRF does not suffer from the extra defense given by same-site cookies. A possible solution would be to not only let the browser enforce the same-site policy for cookies in the `Cookie` header, but also for cookies in the `Set-Cookie` header. This way, it could refuse to save a cookie if it was included in a response to a cross-site request.

It is also important to note that same-site cookies can be bypassed by using subdomains. This results in a rather large attack surface in which the defense due to same-site cookies is mitigated. If it would turn out feasible, we propose to redefine the way same-site requests are defined in the same-site cookie Internet Draft.

In our analysis, we also mentioned smaller inconsistencies like the difference in behavior in case of server-side and client-side requests, but also opening a link in the current or other tab. These can be exploited in small ways, for example by tricking the user to open a link in a new tab. Other than that, users can experience these inconsistencies as annoying.

VII. Related work

Although several defenses against the discussed attacks have been mentioned in this paper, these are all specific to a certain attack. Like same-site cookies, other mechanisms that focus on cross-site requests in general have been developed. It is also important to note that the end-user is the target of these attacks, not so much the web applications that he uses. As a logical consequence, client-side defenses have been developed that can be employed by the end-user. The first one being RequestRodeo, a proxy on the end-user's computer that intercepts every request and strips its implicit authentication if necessary [15]. The rise in popularity of browser extensions for modern web browsers made it easier to manipulate outgoing requests. Numerous extensions have been developed with the intention to give the end-user a means to defend himself better. One of those is CsFire, an extension developed by researchers at KU Leuven [3], [16]. This extension tries to provide a sufficient defense against cross-site attacks, while retaining a satisfactory level of usability. Requests are handled based on policies, composed by the end-user and remotely by the developers.

The policy around same-site cookies is also enforced by the end-user's web browser. However, the web server itself decides the settings of this policy by assigning the `SameSite` attribute. This can be played out as an advantage, as the web server can adjust its cookie strategy around this defense mechanism. To our knowledge, their have not been any other mentions of the same-site cookie functionality in other scientific works. We also believe that this is the first proposed prototype to ease the adoption of same-site cookies.

VIII. Conclusion

This paper discusses a new defense mechanism against cross-site attacks, named the same-site cookie. This defense mechanism is not yet supported by all web browsers and adoption by popular websites is near non-existent. In order to ease adoption of the same-site cookie, we developed an Apache module that gives the website administrator more control in applying the same-site cookie for session cookies. We analysed the functionality of the same-site cookie and found some inconsistencies which could be exploited for malicious intent.

While the same-site cookie appears to provide an appropriate level of in-depth defense against CSRF, XSSI and timing attacks, it remains insufficient against login CSRF. The attack surface to bypass same-site cookies, however, is rather big as requests between subdomains and parent domains are considered same-site. This has a negative effect on all three major attacks it is intended to fend off.

References

[1] OWASP. Owasp top ten project 2013. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[2] W. Zeller and E. W. Felten, "Cross-site request forgeries: Exploitation and prevention," 2008.

[3] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen, "Csfire: Transparent client-side mitigation of malicious cross-domain requests," *Lecture Notes in Computer Science*, pp. 18–34, 2010. [Online]. Available: https://lirias.kuleuven.be/bitstream/123456789/260893/1/paper.pdf

[4] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 75–88. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455782

[5] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," *24th USENIX Security Symposium (USENIX Security 15)*, pp. 723–735, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

[6] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 621–628. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242656

[7] T. V. Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1382–1393, 2015.

[8] M. West and M. Goodwin, "Same-site cookies," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-httpbis-cookie-same-site-00, June 2016. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cookie-same-site-00.txt

[9] C. Bentzel. (2011) Chrome prerendering. [Online]. Available: https://www.chromium.org/developers/design-documents/prerender

[10] A. Barth, "HTTP State Management Mechanism," Internet Requests for Comments, RFC Editor, RFC 6265, April 2011. [Online]. Available: https://tools.ietf.org/html/rfc6265

[11] R. F. Ed. and J. R. Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," Internet Requests for Comments, RFC Editor, RFC 7231, June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7231

[12] M. West, "Cookie prefixes," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-httpbis-cookie-prefixes-00, February 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-httpbis-cookie-prefixes-00

[13] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 413–428. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/huang

[14] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver, "Cookies lack integrity: Real-world implications," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 707–721. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/zheng

[15] M. Johns and J. Winter, "Requestrodeo: client side protection against session riding," in *in Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pp. 5–17.

[16] DistriNet. Csfire browser extensie. [Online]. Available: https://distrinet.cs.kuleuven.be/software/CsFire/

Bibliografie

[1] Adam Barth. *HTTP State Management Mechanism*. RFC 6265. IETF Secretariat, apr 2011, p. 1–37. URL: https://tools.ietf.org/html/rfc6265.

[2] Adam Barth. *The Web Origin Concept*. RFC 6454. IETF Secretariat, dec 2011, p. 1–20. URL: https://tools.ietf.org/html/rfc6454.

[3] Adam Barth, Collin Jackson en John C. Mitchell. "Robust Defenses for Cross-site Request Forgery". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: ACM, 2008, p. 75–88. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455782. URL: http://doi.acm.org/10.1145/1455770.1455782.

[4] Andrew Bortz en Dan Boneh. "Exposing Private Information by Timing Web Applications". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, p. 621–628. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242656. URL: http://doi.acm.org/10.1145/1242572.1242656.

[5] DistriNet. *CsFire browser extensie*. URL: https://distrinet.cs.kuleuven.be/software/CsFire/ (bezocht op 10-04-2017).

[6] Dropbox. *Preventing cross-site attacks using same-site cookies*. URL: https://blogs.dropbox.com/tech/2017/03/preventing-cross-site-attacks-using-same-site-cookies/ (bezocht op 31-05-2017).

[7] R. Fielding Ed. en J. Reschke Ed. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. IETF Secretariat, jun 2014, p. 1–101. URL: https://tools.ietf.org/html/rfc7231.

[8] Jochen Eisinger en Emily Stark. *Referrer Policy*. Candidate Recommendation. W3C, jan 2017. URL: https://www.w3.org/TR/2017/CR-referrer-policy-20170126/.

[9] GitHub. *Yummy cookies across domains*. 2013. URL: https://github.com/blog/1466-yummy-cookies-across-domains (bezocht op 24-05-2017).

[10] Tom Van Goethem, Wouter Joosen en Nick Nikiforakis. "The Clock is Still Ticking: Timing Attacks in the Modern Web". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), p. 1382–1393. DOI: http://dl.acm.org/citation.cfm?doid=2810103.2813632.

[11] Tom Van Goethem e.a. “Request and Conquer: Exposing Cross-Origin Resource Size”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, p. 447–462. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/goethem.

[12] Ilya Grigorik. *Resource Hints*. W3C Working Draft. W3C, mei 2017. URL: https://www.w3.org/TR/resource-hints/.

[13] Google Groups. *Intent to Deprecate and Remove: Prerender*. URL: https://groups.google.com/a/chromium.org/forum/#%21topic/blink-dev/0nSxuuv9bBw (bezocht op 31-05-2017).

[14] Lin-Shung Huang e.a. “Clickjacking: Attacks and Defenses”. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, p. 413–428. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/huang.

[15] Isatou Hydara e.a. “Current state of research on cross-site scripting (XSS) - A systematic literature review”. English. In: deel 58. Complete. 2015, p. 170–186. DOI: 10.1016/j.infsof.2014.07.010.

[16] Martin Johns en Justus Winter. “RequestRodeo: client side protection against session riding”. In: *in Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*. P. 5–17.

[17] Anne van Kesteren. *Cross-Origin Resource Sharing*. W3C Recommendation. W3C, jan 2014. URL: http://www.w3.org/TR/2014/REC-cors-20140116/.

[18] S. Lekies e.a. “The Unexpected Dangers of Dynamic JavaScript”. In: *24th USENIX Security Symposium (USENIX Security 15)* (2015), p. 723–735. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies.

[19] Mozilla Developer Network. *Fetch API*. URL: https://developer.mozilla.org/en/docs/Web/API/Fetch_API (bezocht op 30-05-2017).

[20] Mozilla Developer Network. *HTTP access control (CORS)*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS (bezocht op 23-05-2017).

[21] Mozilla Developer Network. *Inheritance and the prototype chain*. URL: https://developer.mozilla.org/en/docs/Web/JavaScript/Inheritance_and_the_prototype_chain (bezocht op 23-05-2017).

[22] Mozilla Developer Network. *Using the application cache*. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache (bezocht op 30-05-2017).

[23] OWASP. *OWASP Top Ten Project 2013*. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#OWASP_Top_10_for_2013 (bezocht op 08-04-2017).

[24] OWASP. *OWASP Top Ten Project 2017 Release Candidate*. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2017_Release_Candidate (bezocht op 31-05-2017).

[25] Alex Russell e.a. *Service Workers 1*. W3C Working Draft. W3C, okt 2016. URL: https://www.w3.org/TR/2016/WD-service-workers-1-20161011/.

[26] P. De Ryck e.a. "CsFire: Transparent client-side mitigation of malicious cross-domain requests". In: *Lecture Notes in Computer Science* (2010), p. 18–34. URL: https://lirias.kuleuven.be/bitstream/123456789/260893/1/paper.pdf.

[27] Gustav Rydstedt e.a. "Busting frame busting: a study of clickjacking vulnerabilities at popular sites". In: *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*. 2010. URL: https://seclab.stanford.edu/websec/framebusting/framebust.pdf.

[28] S. Schinzel. "An Efficient Mitigation Method for Timing Side Channels on the Web". In: *Proceedings of the 2nd International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2011)*. 2011.

[29] StatCounter. *Browser Market Share Worldwide 2016*. URL: http://gs.statcounter.com/browser-market-share/all/worldwide/2016 (bezocht op 31-05-2017).

[30] Chrome Platform Status. *'Same-site' cookie attribute*. URL: https://www.chromestatus.com/feature/4672634709082112 (bezocht op 11-04-2017).

[31] Takeshi Terada. "Identifier based XSSI attacks". In: mrt 2015. URL: http://www.mbsd.jp/Whitepaper/xssi.pdf.

[32] W3Techs. *Usage of web servers for websites*. Mei 2017. URL: https://w3techs.com/technologies/overview/web_server/all (bezocht op 29-05-2017).

[33] Helen Wang e.a. *The Multi-Principal OS Construction of the Gazelle Web Browser*. Tech. rap. Feb 2009. URL: https://www.microsoft.com/en-us/research/publication/the-multi-principal-os-construction-of-the-gazelle-web-browser/.

[34] Mike West. *Cookie Prefixes*. Internet-Draft draft-ietf-httpbis-cookie-prefixes-00. IETF Secretariat, feb 2016. URL: https://tools.ietf.org/html/draft-ietf-httpbis-cookie-prefixes-00.

[35] Mike West. *First-Party Cookies*. Internet-Draft draft-west-first-party-cookies-00. IETF Secretariat, okt 2014. URL: https://tools.ietf.org/html/draft-west-first-party-cookies-00.

[36] Mike West en Mark Goodwin. *Same-Site Cookies*. Internet-Draft draft-ietf-httpbis-cookie-same-site-00. IETF Secretariat, jun 2016. URL: http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cookie-same-site-00.txt.

[37] WHATWG. *Storage*. Mei 2017. URL: https://storage.spec.whatwg.org/ (bezocht op 24-05-2017).

[38] WHATWG. *XMLHttpRequest*. Mei 2017. URL: https://xhr.spec.whatwg.org/ (bezocht op 06-06-2017).

[39] Kinuko Yasuda. *Quota Management API*. Tech. rap. W3C, mei 2016. URL: http://www.w3.org/TR/2016/NOTE-quota-api-20160523/.

[40] William Zeller en Edward W. Felten. *Cross-Site Request Forgeries: Exploitation and Prevention*. 2008.

[41] Xiaofeng Zheng e.a. "Cookies Lack Integrity: Real-World Implications". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, p. 707–721. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/zheng.

Fiche masterproef

Student: Gertjan Franken

Titel: Struikelblokken en kwetsbaarheden bij de adoptie van same-site cookies

Engelse titel: Obstacles and vulnerabilities concerning the adoption of same-site cookies

UDC: 681.3

Korte inhoud:

Cross-site aanvallen zoals *cross-site request forgery*, *cross-site script inclusion* en *cross-site timing attacks* vormen nog altijd een prominente dreiging op het web. Ondanks de verschillende verdedigingsmechanismen hiertegen, worden er nog altijd cruciale kwetsbaarheden vastgesteld bij populaire websites. Sinds kort ondersteunen de webbrowsers Google Chrome en Opera een nieuw verdedigingsmechanisme tegen deze aanvallen, namelijk de same-site cookie. Helaas gebeurt de adoptie van deze nieuwe vorm van bescherming traag, daar ook niet alle populaire webbrowser ondersteuning bieden. Hierdoor is er ook nog niet veel bekend over het gebruik, de werking en de effectiviteit van same-site cookies. In deze thesis verrichten we verkennend onderzoek naar same-site cookies. We overlopen de eigenlijke werking van dit nieuwe concept en verhelderen enkele onduidelijkheden aanwezig in de Internet Draft van de same-site cookie. Dit doen we door zelf een gedetailleerde analyse uit te voeren met als doel het gedrag van same-site cookies in verschillende scenario's te achterhalen. Via dit verkennend onderzoek ontdekten we een bug en enkele inconsistenties, die we dan ook bespreken. We stellen ook een prototype voor dat de adoptie van de same-site cookie tracht te vergemakkelijken. Dit prototype is een uitbreidingsmodule voor webservers en kan voor verschillende soorten webservers geïmplementeerd worden. Ten slotte evalueren we dit prototype in combinatie met same-site cookies op basis van performantie en veiligheid tegen bestaande aanvallen. We tonen aan dat, ondanks de gevonden inconsistenties, same-site cookies een degelijk middel zijn om websites te voorzien van een extra laag van bescherming tegen cross-site aanvallen.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotor: Prof. dr. ir. Wouter Joosen

Assessoren: Dr. S. Michiels
Dr. M. Vanhoef

Begeleider: T. Van Goethem