KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

Veilige integratie van JavaScript advertisements door middel van Secure ECMAScript

Lynsey Hens
Brenda Lissens

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

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

Assessoren:
Prof. dr. ir. H. Blockeel
Dr. P. Philippaerts

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

Academiejaar 2011 – 2012

Voorwoord

Vele mensen hebben geholpen bij de opbouw van deze thesis. We willen dan ook gebruik maken van deze gelegenheid om hen te bedanken. Om te beginnen bedanken we onze begeleiders, Steven Van Acker en Philippe De Ryck die ons met hun kennis hebben bijgestaan tijdens het hele proces. Daarnaast bedanken we ook onze promotoren, Frank Piessens en Lieven Desmet, die er voor gezorgd hebben dat deze thesis de juiste richting insloeg.

We willen ook zeker Guy, Roel en Magriet bedanken voor het nalezen van de thesistekst samen met onze ouders en zussen voor de permanente steun. Als laatste vergeten we ook niet onze vrienden die er waren met hun steun wanneer het niet allemaal van een leien dakje liep.

Lynsey Hens
Brenda Lissens

Inhoudsopgave

Samenvatting

Vele inkomsten in de online wereld worden gegenereerd door het opnemen van advertenties in websites. Deze advertenties worden met behulp van JavaScript geïntegreerd in de pagina van de publiceerder. Deze laatste heeft echter geen idee over wat deze scripts juist zullen uitvoeren binnen de context van zijn webpagina. Deze kunnen dus ook evengoed kwaadaardige intenties hebben waardoor bijvoorbeeld persoonlijke informatie dreigt vrijgegeven te worden. De bedoeling van deze thesis is om deze scripts op een gecontroleerde manier uit te voeren zodat de publiceerder geen veiligheidsrisico's meer loopt. Er bestaan al heel wat technieken om dit doel te verwezenlijken, elk met hun eigen voor- en nadelen. In deze thesis hebben we geprobeerd om een bestaande oplossing, namelijk WebJail, te verbeteren. Bij WebJail moest de browser immers intern gewijzigd worden om te kunnen voldoen aan de vereiste veiligheid. Hierdoor was er gebruikersondersteuning nodig wat een enorm nadeel vormt. Met behulp van SecureECMAScript hebben we een prototype kunnen ontwikkelen waarbij browserwijzigingen niet langer nodig zijn. Tot slot vergelijken we het ontwikkelde prototype met het prototype gebaseerd op de bestaande techniek AdJail.

Lijst van figuren en tabellen

Lijst van figuren

Lijst van tabellen

Lijst van afkortingen

Afkortingen

SES	Secure EcmaScript
HTML	HyperText Markup Language

Hoofdstuk 1

Inleiding

Vele webpaginabeheerders genereren inkomsten door advertenties van externe partijen, bijvoorbeeld advertentienetwerken, op te nemen in hun website. Vele van deze advertenties zijn toegespitst op de inhoud van de webpagina. Op die manier worden bezoekers meer aangetrokken om op deze advertenties de klikken. Dit heet *contextual advertising*. Verschillende advertentienetwerken voorzien advertisement scripts om deze advertenties correct weer te geven. De beheerder heeft echter geen weet over wat zo een script nu juist zal uitvoeren. Dit kan dus evengoed een script met kwaadaardige intenties zijn waardoor bijvoorbeeld persoonlijke informatie dreigt vrijgegeven te worden. De bedoeling van deze thesis is om advertisement scripts die gebruik maken van JavaScript, gecontroleerd uit te voeren en contextual advertising toch mogelijk te maken. We willen dat het script niet meer, maar ook niet minder functionaliteit ter beschikking heeft dan nodig is. Binnen dit domein zijn al heel wat technieken ontworpen om dit doel te ondersteunen. Elk van deze technieken hebben voordelen en nadelen. Zelf zullen we een prototype uitwerken dat het script zal isoleren met behulp van Secure ECMAScript [2].

In het tweede hoofdstuk worden eerst de verschillende technieken verder uitgediept, voorafgaand geven we nog een algemene schets van enkele basisconcepten die doorheen de tekst aan bod komen. In hoofdstuk 3 stellen we het ontworpen prototype voor. Om de vergelijking te kunnen maken met andere technieken werken we ook een prototype uit dat gebaseerd is op een reeds bestaande techniek, namelijk AdJail [24]. Dit wordt uitgebreid beschreven in hoofdstuk 4. Op basis hiervan kunnen we dan achteraf conclusies trekken over de bruikbaarheid en de veiligheid van het ontworpen prototype alsook over de voor- en nadelen van de gebruikte isolatiemanieren. Hoofdstuk 5 behandelt deze vergelijking. We besluiten met een beeld te vormen over de resultaten van het gevoerde onderzoek.

Hoofdstuk 2

Achtergrond

In het huidige internettijdperk zijn allerhande toepassingen beschikbaar voor dagelijks gebruik. Deze brengen echter gevaren met zich mee waar de gebruiker zich niet altijd bewust van is. In dit hoofdstuk zullen we voornamelijk dieper ingaan op het gebruik van mashups, wat deze toepassingen voorstellen, hoe deze op dit moment meestal geïmplementeerd worden en met welke risico's deze manieren gepaard gaan. We geven een kort overzicht over mogelijke oplossingen voor deze risico's samen met hun voor- en nadelen. Vervolgens bespreken we een specifiek voorbeeld van mashups nl. advertenties. We eindigen dit hoofdstuk met de probleemstelling van deze thesis.

2.1 Mashup

2.1.1 Algemeen concept

Het algemeen concept van een mashup is het samenvoegen van twee of meerdere componenten tot één functioneel geheel. Mashups komen in verschillende domeinen aan bod. Zo komt het begrip mashup origineel uit de muziek waarbij twee of meerdere nummers door elkaar gemengd worden.

Ook binnen het domein van de informatica komen mashups aan bod. Hierbij stelt de mashup een applicatie voor die andere bestaande toepassingen samenvoegt tot één geheel, mogelijk extra functionaliteit toevoegt en hierdoor een meerwaarde biedt voor de gebruiker. Web mashups bijvoorbeeld zijn webpagina's die componenten van bestaande online toepassingen gaan opnemen in hun pagina. Het samenwerken van de verschillende onderdelen van de mashup is van cruciaal belang voor de bruikbaarheid van de mashup.

Een uitgesproken voorbeeld van een web mashup is *i*Google [12]. *i*Google laat de gebruiker toe naar eigen voorkeur toepassingen toe te voegen zoals Facebook [1], Twitter [3], Google Maps [11], etc. Doordat deze applicaties vanuit éénzelfde pagina toegankelijk zijn en er een overzicht geboden wordt, levert dit een bijdrage aan het gebruiksgemak van de verschillende onderdelen. Daarnaast is het ook mogelijk dat bepaalde informatie van de ene component kan gebruikt worden in een andere

Figuur 2.1: *i*Google met Google Maps, Twitter en Facebook

component. In figuur 2.1 wordt een voorbeeld van *i*Google geïllustreerd.

Er schuilen natuurlijk ook gevaren in het gebruik van mashups. Uit het feit dat componenten met elkaar kunnen interageren en communiceren, volgt ook dat de verschillende componenten elkaar kunnen beïnvloeden. Daarnaast heeft een component van de mashup toegang tot de data en resources van de omhullende pagina (bv. cookies) en de data van de andere componenten die deel uitmaken van dezelfde mashup, ook wanneer hij deze niet nodig heeft voor het uitvoeren van zijn kerntaak. Het grote probleem bij mashups is dat ze code van een derde partij ongecontroleerd uitvoeren zonder te weten wat de gevolgen zullen zijn. Door bijvoorbeeld kwaadwillig gebruik van een component kan een groot deel van de informatie gelekt worden of kan er schade aangericht worden door krachtige operaties verkeerd te gebruiken. Om veiligheid te garanderen zouden de mashup componenten geïsoleerd moeten worden zodat ze niet langer toegang hebben tot cruciale informatie of resources. Hierdoor zou er echter functionaliteit verloren gaan. Er moet dus een compromis gevonden worden tussen de interactie van de verschillende componenten met elkaar en de hoofdpagina enerzijds en isolatie van diezelfde componenten anderzijds.

2.1.2 Huidige implementatietechnieken en hun problemen

De implementatie van mashups gebeurt op dit moment meestal met behulp van iframe-integratie of door middel van scriptinclusie. Beide worden in volgende paragrafen verder uitgediept.

Iframe-integratie

Een eerste manier om een mashup te bouwen, is met behulp van iframe-integratie. Door gebruik te maken van de HTML iframe tag kan een externe pagina in de pagina geladen worden. Hierdoor wordt de code uitgevoerd in een aparte pagina

waaruit volgt dat de desbetreffende component onderhevig is aan het Same Origin Policy principe. Het Same Origin Policy principe is een veiligheidsmechanisme dat afgedwongen wordt door de browser. Dit stelt dat data en resources toegankelijk zijn voor pagina's van dezelfde oorsprong maar afgeschermd worden van pagina's met een verschillende afkomst. Deze afkomst wordt gedefinieerd door de domeinnaam, het gebruikte protocol en de poort. Door een externe component te laden in een iframe, is de informatie van de omhullende pagina dus niet langer toegankelijk voor de geïmporteerde component daar het van een andere oorsprong is.

Deze manier van werken zorgt er echter voor dat de component volledig geïsoleerd is in het iframe waardoor het helemaal geen toegang meer heeft tot de hoofdpagina. Iframe-integratie kan soms te strikt zijn wanneer een ingeladen component toch nog een beperkte hoeveelheid informatie nodig heeft of wanneer interactie tussen verschillende componenten nodig is. Daarnaast belet deze aanpak niet dat alle JavaScript operaties toch nog uitgevoerd kunnen worden, al is het binnen de context van het iframe. Het opvragen van de huidige locatie kan bijvoorbeeld nog steeds ondanks de isolatie in een iframe. Het aangehaalde mashup voorbeeld *i*Google is gebaseerd op iframe-integratie.

Script Inclusie

Als volgende manier voor het bouwen van mashups komt script inclusie aan bod. Met éénvoudige script tags wordt de externe code geïmporteerd in de hoofdpagina. In tegenstelling tot iframe-integratie, wordt bij script inclusie de geïmporteerde code uitgevoerd in dezelfde context als de omhullende pagina. Van het Same Origin Policy principe is hier dus geen sprake. Op de componenten worden weinig restricties opgelegd waardoor deze toegang heeft tot alle cruciale data en resources van de omhullende pagina. De hoofdpagina kan willekeurig aangepast worden door de externe code en is onderhevig aan grote beveiligingsrisico's.

2.1.3 Oplossingen

Least-privilege principe

De ideale oplossing die zowel voldoende isolatie oplegt en toch de gevraagde functionaliteit en interactie blijft aanbieden, wordt samengevat door het least-privilege principe. Dit principe stelt dat elke component enkel en alleen die rechten heeft om zijn kernfunctionaliteit te kunnen uitvoeren. De component heeft nog steeds alle operaties, die nodig zijn voor zijn kernfunctionaliteit, ter beschikking. Maar omdat elke component alleen die operaties kan uitvoeren die hij nodig heeft, is er geen risico meer dat hij ongeoorloofd andere wijzigingen gaat doorvoeren. Interactie en communicatie tussen de verschillende componenten van de mashup is dus nog steeds mogelijk. Op deze manier verliest de mashup noch aan functionaliteit noch aan veiligheid. De vorige implementatietechnieken nl. iframe-integratie en script inclusie, voldoen beide niet aan het least-privilege principe. Iframe-intergratie is op sommige vlakken te strikt terwijl script inclusie geen isolatie van de componenten

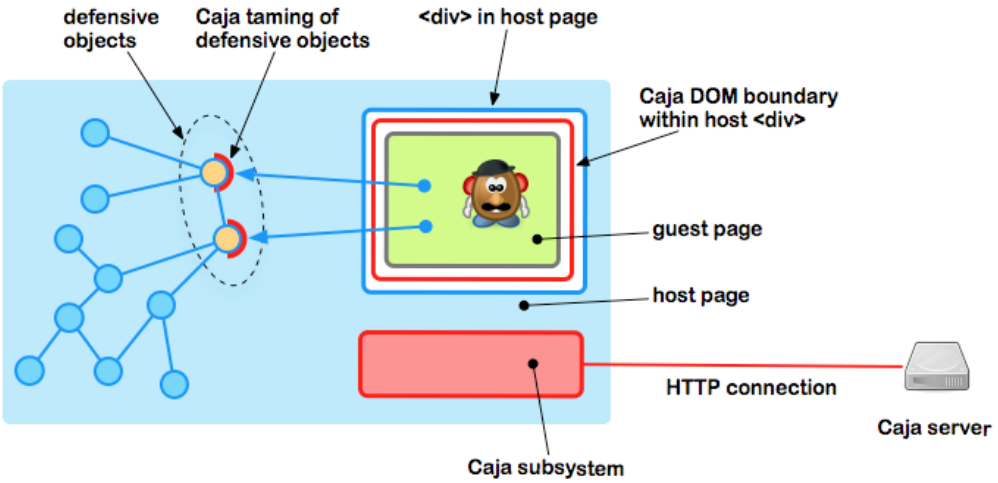
aanbiedt.

In de context van mashups zijn er al enkele technieken ontwikkeld die gebaseerd zijn op het least-privilege principe. In volgende paragrafen geven we een kort overzicht over deze bestaande aanpakken.

Caja

Caja [10, 20] is een techniek die verder bouwt op script inclusie. Caja realiseert het least-privilege principe door ervoor te zorgen dat alleen maar een veilige subset van JavaScript gebruikt kan worden. Deze subset stelt een Capability Security Language voor. Het principe van Capability Security Language houdt in dat een object alleen maar toegang heeft tot een ander object als het een referentie heeft naar dat object. De communicatie met en de toegang tot objecten gebeurt door het geven van referenties naar die objecten. Deze veilige subset wordt gemaakt door de externe code eerst te transformeren ("cajolen") en dan pas in de mashup op te nemen.

Figuur 2.2 stelt dit concept visueel voor. De gecajoled code vormt een "sandbox" die standaard geen toegang heeft tot de omhullende pagina. Een sandbox is een ruimte waarin code geïsoleerd kan uitgevoerd worden. Enkel via referenties (defensieve objecten) die meegegeven zijn door de mashup integrator heeft de geïsoleerde code toegang tot de hoofdpagina.



Figuur 2.2: Onderdelen van Caja [10].

Caja neemt isolatie als standaardwaarde. Op die manier gebeurt de interactie met de hoofdpagina op een sterk gecontroleerde manier en worden er minder snel veiligheidsgevoelige functies over het hoofd gezien. De omhullende pagina hoeft alleen die referenties aan de gastpagina te geven die nodig zijn om te kunnen functioneren.

Een nadeel van Caja is dat een component door Caja altijd eerst gecajoled moet worden voordat het gebruikt mag worden in de mashup. Hierdoor ontstaat er een heel dichte koppeling tussen de beveiliging en de externe code. Er worden ook beperkingen opgelegd op het gebruik van bepaalde JavaScript methoden en objecten. Bijvoorbeeld het gebruik van `eval`, `with`, `caller`, `callee`, `__defineGetter__`, ... is niet toegelaten. Wanneer dit wel gebruikt wordt in de gastpagina, kan deze niet gecajoled worden en dus niet gebruikt worden met Caja.

Om de code te kunnen transformeren, moet de mashup integrator in het bezit zijn van de JavaScript code. Dit is niet altijd vanzelfsprekend en vormt dus een bijkomende nadeel.

Secure ECMAScript

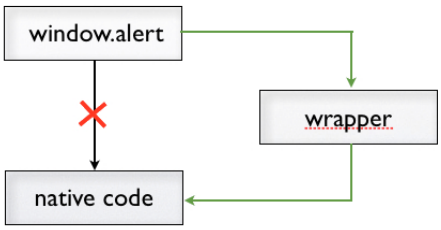
Kort samengevat is Secure ECMAScript [2] de client-side versie van Caja. Net zoals Caja probeert Secure ECMAScript er voor te zorgen dat de geïmporteerde code wordt omgevormd tot een Capability Security Language waarbij communicatie tussen twee objecten enkel kan plaatsvinden wanneer er een referentie bestaat tussen deze 2 objecten.

Met behulp van een aantal JavaScript bibliotheken die beschikbaar worden gesteld, wordt de geïmporteerde code client-side getransformeerd naar een veilige versie. Dit in tegenstelling tot server-side Caja waarbij de code naar een Cajoler op een externe server wordt gestuurd.

De geïmporteerde code wordt volledig gesandboxt zodat deze code standaard aan geen enkele operatie aan kan. Slechts wanneer operaties meegegeven worden in een policy aan de client-side Caja compiler, zal de gesandboxte code deze kunnen gebruiken maar enkel via deze beschermde operaties. Op die manier kan men op eenvoudige wijze restricties opleggen aan de geïmporteerde code.

Self-Protecting JavaScript

Self-protecting JavaScript [22, 23, 17] is een techniek die, net zoals Caja, gebaseerd is op script inclusie. Bij deze techniek wordt er in de header van de webpagina een extra JavaScript bestand toegevoegd die wrappers plaatsen rond de native JavaScript functies. Op deze manier kunnen er policies opgelegd worden op built-in operaties en kan de uitvoering ervan gecontroleerd worden. De originele functie wordt dus vervangen door een JavaScript wrapper die als enige een referentie bevat naar de werkelijke built-in functionaliteit. In deze wrapper zit de policy vervat en enkel wanneer er aan de policy voldaan is, zal de wrapper toelaten de oorspronkelijke functionaliteit op te roepen. Hoe dit concreet in zijn werk gaat, wordt met een voorbeeld geïllustreerd in figuur 2.3.

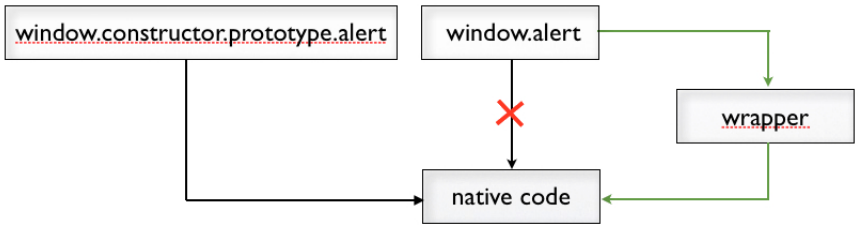


Figuur 2.3: Voorbeeld van een wrapper.

De wrap-functionaliteit wordt, zoals eerder vermeld, in de header van de webpagina toegevoegd. Hierdoor is het wrappen van de te beveiligen functies het eerste wat op de webpagina uitgevoerd wordt en is het niet mogelijk dat externe code nog voor het wrappen, een referentie bemachtigt naar de originele implementatie.

Self-Protecting JavaScript heeft als voordeel dat het een lichtgewicht aanpak is. Het is namelijk niet nodig om code te transformeren en er worden ook geen beperkingen opgelegd aan het gebruik van JavaScript. Self-Protecting JavaScript biedt dus de mogelijkheid aan om policies op te leggen aan built-in JavaScript functies. Daar dit wrappen echter op een hoog niveau aangeboden wordt, blijven er enkele belangrijke nadelen aanwezig.

Eén van de belangrijkste nadelen van Self-Protecting JavaScript is dat de wrapper slechts één alias naar de originele functie wrapt. Zo kan de native functie 'alert' rechtstreeks aangeroepen worden op het window-object maar kan het ook uitgevoerd worden door langs de constructor en het prototype te gaan. Wanneer echter window.alert gewrapt wordt, blijft het pad via de constructor en het prototype onbeveiligd. Figuur 2.4 toont hoe de wrapper op deze manier omzeild kan worden.



Figuur 2.4: Voorbeeld waarbij de wrapper omzeild wordt.

WebJail

De architectuur van WebJail bouwt verder op iframe-integratie. Door het gebruik van iframe-integratie kan vertrouwd worden op de bescherming aangeboden door het Same Origin Policy principe. Zoals eerder vermeld, bood dit echter niet de vereiste bescherming. De data van de hoofdpagina wordt wel afgeschermd maar alle JavaScript operaties zijn nog steeds toegankelijk voor de iframe component, al is het dan binnen de context van het iframe. WebJail komt tegemoet aan de tekortkoming wat betreft veiligheid wanneer enkel gebruik gemaakt wordt van iframe integratie.

Het idee achter WebJail stelt dat de mashup integrator op elke component die hij importeert, een policy kan opleggen. Deze policy bepaalt wat de component wel en niet mag. Elke component wordt ingeladen in een iframe, via een policy attribuut van het iframe element wordt deze policy gekoppeld aan de desbetreffende component. Elk van deze hoog level policies wordt omgezet naar adviesfuncties voor elke JavaScript operatie afzonderlijk. Deze adviesfuncties zijn eveneens in JavaScript opgemaakt. Deze functie zal in overeenstemming met de policy eerst verifiëren of de aanroep naar een gevoelige operatie toegestaan is en pas daarna de oorspronkelijke implementatie uitvoeren.

Het is echter mogelijk dat er al referenties naar de oorspronkelijke implementatie verkregen zijn vooraleer de adviesfunctie geregisteerd werd. Om te verzekeren dat elke aanroep naar de oorspronkelijke built-in functies toch via de bijhorende adviesfunctie plaats vindt en dus de policy niet omzeild kan worden, is **deep aspect weaving** nodig. De implementatie is gebaseerd op het principe van ConScript [18]. De C++ implementatie van de oorspronkelijke functie wordt op een laag niveau, in de browser, vervangen door de adviesfunctie. Doordat de adviesfunctie vanaf dan de enige is met een referentie naar de originele functie, is het onmogelijk om de policy te omzeilen.

Deze manier van werken verzekert dat het onmogelijk is voor een aanvaller een operatie uit te voeren die niet voldoet aan de policy. Deze methode brengt echter ook een nadeel met zich mee. De implementatie van de originele functies moet, zoals eerder vermeld, in de browser gewijzigd worden. Dit zorgt er voor dat WebJail gebruikersondersteuning vereist en dat er bij eventuele wijzigingen updates geïnstalleerd zullen moeten worden.

2.2 Advertenties

2.2.1 Algemeen concept

Een specifiek voorbeeld van web mashups is het integreren van advertenties in webpagina's. Een groot deel van de huidige inkomsten van grote marktspelers zoals bijvoorbeeld Google komt van advertenties die ze afbeelden op hun pagina's. Publiceerders nemen advertenties op in hun webpagina en worden vergoed wanneer hun bezoekers klikken op een weergegeven advertentie. Deze advertenties worden aange-

boden door adverteerders of door advertentienetwerken. Enkele bekende voorbeelden van deze advertentienetwerken zijn Google AdSense [14] en Microsoft AdCenter [19].

Een belangrijk concept in deze advertisement cyclus is *targeting*. Targeting zorgt ervoor dat de getoonde advertentie afgestemd is op de bezoeker van de pagina. Aangezien hierdoor de advertentie meer aansluit bij de interesses van de webgebruiker, vergroot dit de kans dat hij effectief op de advertentie zal klikken. Targeting wordt mogelijk gemaakt door het gebruik van advertisement scripts. Dit is third party code, die op basis van de inhoud van onder andere de huidige pagina, zal bepalen welke advertenties van het advertentienetwerk geschikt zijn om weer te geven. Contextual advertising is een categorie binnen targeting. Hierbij scant het advertisement script de webpagina op zoek naar kernwoorden om op basis hiervan te bepalen welke advertenties gerelateerd zijn aan de pagina-inhoud.

Deze advertisement scripts, verantwoordelijk voor het afstemmen van de advertenties op de gebruiker, zijn afkomstig van externe partijen. Hierdoor weet de publiceerder, degene die de advertenties opneemt in zijn pagina, niet of deze derde partij vertrouwd kan worden. Daar de JavaScript code van het advertisement script aan de context van de browser en aan de inhoud van de pagina kan, heeft het script de toegang tot heel wat confidentiële informatie. Wanneer deze externe partij kwaadwillige intenties heeft, kan dit script persoonlijke gegevens vrijgeven, de bezochte pagina of applicatie wijzigen, onbevoegd acties in de browser uitvoeren en zelfs via zwakheden in de browser het gebruikerssysteem van de webbezoeker overnemen.

2.2.2 Huidige implementatietechnieken en hun problemen

Net zoals bij de implementatie van mashups in het algemeen zijn iframe integratie en script inclusie de meest voorkomende manieren voor het opnemen van advertenties. Ook hier komen dezelfde problemen bovendrijven. Bij iframe integratie wordt de component volledig geïsoleerd waardoor contextual advertising onmogelijk wordt. Script inclusie daarentegen legt geen enkele restrictie op waardoor de externe pagina alle mogelijke operaties ongecontroleerd kan uitvoeren.

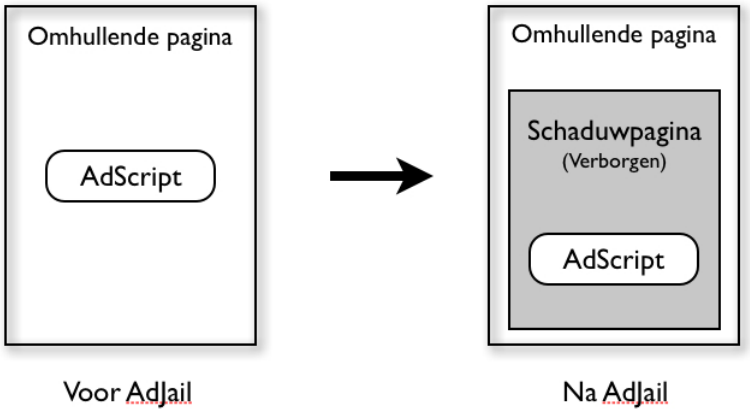
2.2.3 Oplossingen

Om advertenties op een geïsoleerde manier op te nemen in de integrerende pagina zonder verlies aan interactie, grijpen we terug naar het least-privilege principe (zie 2.1.3) waarbij het advertisement script enkel toegang krijgt tot de JavaScript operaties die het nodig heeft voor zijn kernfunctionaliteit. Naast de reeds aangereikte oplossingen voor mashups in het algemeen, zijn er ook een aantal technieken die zich specifiek richten op mashups met advertenties.

AdJail

Het hele concept van advertenties geïntegreerd in een webpagina, is gebaseerd op scripts die de gepaste advertenties selecteren op basis van informatie aanwezig op

de hoofdpagina (*targeting*). AdJail [24] verhoogt de veiligheid door deze scripts en hun advertenties te isoleren in een schaduwpagina. Deze schaduwpagina wordt onzichtbaar opgenomen in de hoofdpagina met behulp van een iframe. Op deze manier wordt gesteund op het Same Origin Policy principe dat afgedwongen wordt door de browser en kan het script van de advertentie niet meer onbeperkt aan de gegevens aanwezig op de omhullende pagina. Dit principe wordt geïllustreerd in figuur 2.5



FIGUUR 2.5: Het advertisment script wordt verplaatst naar een verborgen schaduwpagina. [24]

Om de advertentie toch op een degelijke manier te kunnen laten werken is het nodig dat het script gecontroleerde toegang heeft tot de integrerende pagina. Daarom wordt de inhoud van de omhullende pagina die gebruikt mag worden door het script, naar de schaduwpagina gestuurd met behulp van interframe communicatie.

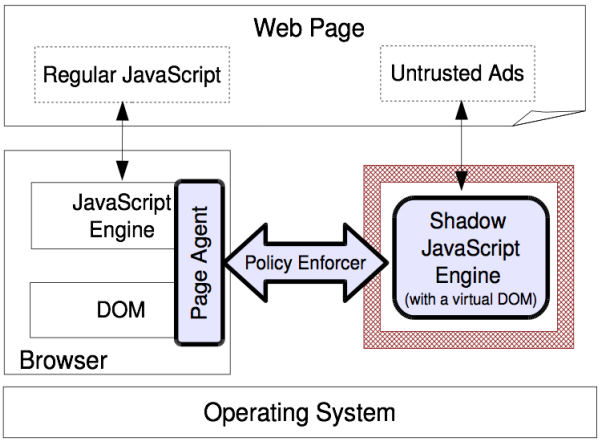
De publiceerder van de advertenties behoudt alle controle over de werking van de advertenties met behulp van policies. Elk element aanwezig op een HTML pagina kan geannoteerd worden met voorgedefinieerde policies. Deze bepalen onder andere welke elementen van de integrerende pagina gelezen mogen worden door een advertisement script, naar welke elementen een advertisement script kan schrijven, of de inhoud van een advertentie afbeeldingen mag bevatten, etc. Deze policies worden afgedwongen op elke inhoud die tussen de 2 pagina's uitgewisseld wordt.

AdJail biedt op deze manier bescherming tegen het lekken van vertrouwelijke informatie en het uitvoeren van andere schadelijke JavaScript code.

AdSentry

Net zoals bij AdJail kan de publiceerder bij AdSentry [9] policies opleggen aan het advertisement script die bepalen tot wat het script toegang heeft. AdSentry biedt isolatie van de advertisement script door deze scripts uit te voeren in een shadow JavaScript engine. Op deze manier hebben onvertrouwde advertisement scripts geen invloed op de integrerende webapplicatie zonder controle van de publiceerder.

AdSentry werkt dus met twee JavaScript engines. De scripts van de omhullende pagina worden uitgevoerd in de standaard engine terwijl de geïmporteerde scripts uitgevoerd worden in een aparte shadow engine. Om te kunnen bepalen welke advertenties geschikt zijn voor de bezoeker, heeft een advertisement script gecontroleerde toegang nodig tot de oorspronkelijke pagina. Daarom biedt AdSentry een **virtueel DOM** aan aan deze shadow engine. Alle paginatoegangen van het advertisement script worden door dit virtueel DOM via een policy enforcer doorgegeven aan de integrerende pagina. De policy enforcer controleert op basis van de opgelegde policies of de operatie mag uitgevoerd worden. Wanneer dit het geval is, wordt de operatie doorgegeven aan de page agent. De page agent zal de operatie uitvoeren in de context van de omhullende pagina en het resultaat terug sturen naar het advertisement script in de shadow JavaScript engine. De architectuur van AdSentry wordt visueel voorgesteld in figuur 2.6.



Figuur 2.6: AdSentry Architectuur [9]

AdSentry heeft enkele belangrijke voordelen ten opzichte van AdJail. Zo biedt AdSentry naast beveiliging tegen het vrijgeven van confidentiële informatie ook een tegenmaatregel tegen zwakheden in de browser waardoor onder andere drive-by-download aanvallen kunnen afgeweerd worden. Doordat in AdJail gebruik gemaakt wordt van een iframe, geeft dit meer problemen met het correct genereren van advertentie impressies (request naar de server van de adverteerder). Dit probleem

komt niet voor bij AdSentry. Bij AdJail moeten de policies ook vooraf gekend zijn. In AdSentry kan dit at runtime gewijzigd worden, door zowel de publiceerder als door de gebruikers zelf.

AdSentry wordt echter geïmplementeerd als een browserextensie waardoor er toch nog een zekere ondersteuning van de gebruiker nodig is. Dit is niet het geval bij AdJail wat in het voordeel van deze laatste spreekt.

2.3 Probleemstelling

WebJail slaagt erin om het least-privilege principe toe te passen maar maakt gebruik van het principe van Deep Aspect Weaving. Dit heeft als voordeel dat het onmogelijk is de adviesfunctie te omzeilen maar om dit te bekomen, moeten er wijzigingen aangebracht worden in de browser waardoor dus impliciet ondersteuning van de gebruiker nodig is. Wanneer de implementatie van WebJail wijzigt, zal de gebruiker hier actief moeten op reageren door eventueel updates te installeren. De huidige client-side implementatie van WebJail vervangen door een server-side uitwerking is de belangrijkste doelstelling van deze thesis waarbij het least-privilege principe behouden blijft. Dit doel gecombineerd met de oorspronkelijke implementatie van WebJail resulteert in de volgende doelstellingen:

Doelstelling 1 De client mag niet gewijzigd worden. WebJail wordt vanuit de server gepusht naar de client. De client hoeft zelf geen actie te ondernemen. Het is niet de bedoeling dat het afdwingen van de policies zelf gecontroleerd wordt aan de server kant. Dit zal nog steeds client-side gebeuren maar zonder extra ondersteuning van de gebruiker.

Doelstelling 2 De mashup integrator kan op elke component een policy opleggen die omschrijft wat de component wel en niet mag. Concreet wil dit zeggen dat de policy vastlegt tot welke operaties de component toegang heeft.

Doelstelling 3 De mashup integrator heeft de verantwoordelijkheid voor het afdwingen van de policies op de componenten. De component moet dus zelf geen specifieke ondersteuning aanbieden om deel te kunnen uitmaken van de mashup (bijvoorbeeld beperkte JavaScript set).

Doelstelling 4 De attacker code die eventueel in de component aanwezig is, kan de beveiliging niet omzeilen. De built-in functies kunnen niet uitgevoerd worden zonder dat de desbetreffende policies er op worden afgedwongen.

We zullen een prototype ontwikkelen dat het idee van WebJail specifiek toepast op advertenties met contextual advertising. Steunend op het principe van Secure EcmaScript [2] kunnen we de browserondersteuning uit WebJail wegwerken.

Hoofdstuk 3

WebJail prototype gebaseerd op Secure EcmaScript

Het WebJail prototype dat geen ondersteuning meer nodig heeft van de gebruiker, zal gebaseerd worden op Secure EcmaScript [2]. Oorspronkelijk was het de bedoeling om de server-side versie van WebJail [7] te baseren op Caja [10]. Omdat de componenten van Caja echter sterk aan elkaar gekoppeld zijn, zou het moeilijk worden enkel die componenten te extracten die nodig waren voor het WebJail concept. Secure EcmaScript biedt, net zoals Caja, de mogelijkheid JavaScript code te isoleren in een sandbox door deze code client-side te transformeren naar een Capability Security Language en is dus een volwaardig alternatief.

3.1 Origineel WebJail Concept

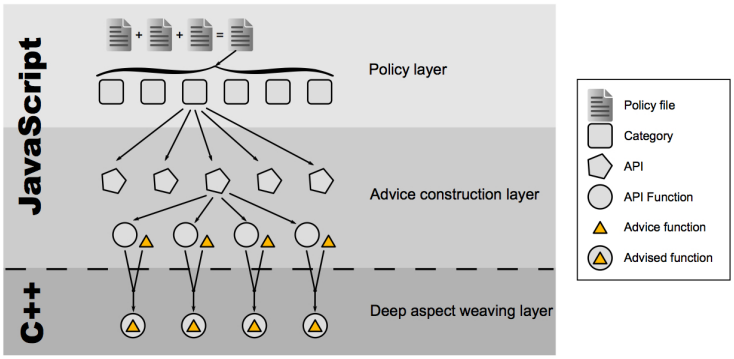
Om een duidelijk beeld te kunnen vormen van waar het Secure EcmaScript prototype naartoe leidt, geven we nog een uitgebreide beschrijving van de originele WebJail implementatie.

3.1.1 Architectuur

WebJail verwezenlijkt het least-privilege principe doordat de mashup integrator met behulp van policies beperkingen opleggen aan de third party component. Deze techniek bouwt verder op iframe-integratie en steunt dus op het same origin policy principe. WebJail bestaat uit drie lagen met elk hun eigen verantwoordelijkheden:

- Policy layer
- Advice construction layer
- Deep aspect weaving layer

Figuur 3.1 geeft een overzicht van deze lagen.



Figuur 3.1: Overzicht van de layers binnen WebJail [7]

Policy layer

De operaties die veiligheidsrisico's inhouden, worden opgedeeld in categorieën zoals o.a. DOM access, cookies, Media, etc. In de policy wordt voor elke categorie opgelijst of de component volledige, beperkte of geen toegang heeft tot de operaties die behoren tot die categorie. De koppeling van de policy aan de bijhorende component gebeurt in de **policy layer**. Figuur 3.2 geeft een overzicht van de verschillende categoriën.

Categories and APIs (# op.)	Whitelist
DOM Access	ElemReadSet, ElemWriteSet
DOM Core (17)	
Cookies	KeyReadSet, KeyWriteSet
cookies (2)	
External Communication	DestinationDomainSet
XHR, CORS, UMP (4)	
WebSockets (5)	
Server-sent events (2)	
Inter-frame Communication	DestinationDomainSet
Web Messaging (3)	
Client-side Storage	KeyReadSet, KeyWriteSet
Web Storage (5)	
IndexedDB (16)	
File API (4)	
File API: Dir. and Syst. (11)	
File API: Writer (3)	
UI and Rendering	
History API (4)	
Drag/Drop events (3)	
Media	
Media Capture API (3)	
Geolocation	
Geolocation API (2)	
Device Access	SensorReadSet
System Information API (2)	
Total number of security-sensitive operations: 86	

Figuur 3.2: Overzicht van de veiligheidsgevoelige operaties .[7]

- *DOM Access:* legt vast welke elementen van de webpagina gelezen en gewijzigd mogen worden.
- *Cookies:* legt vast welke cookies gelezen mogen worden of naar welke cookies geschreven mag worden.

- *Inter-frame communication:* bepaalt of het toegestaan is berichten te versturen tussen twee verschillende webpagina's (bijvoorbeeld tussen twee componenten).
- *Client-side storage:* staat de applicatie toe om data tijdelijk of permanent op te slaan.
- *External communication:* legt vast of communicatie met remote websites mogelijk moet zijn.
- *Device access:* staat de web applicatie toe om systeeminformatie zoals batterij-status, etc. op te vragen.
- *Media:* bepaalt of de applicatie audio en video mag afspelen/opnemen.
- *Geolocation:* legt vast of de geolocatie mag opgevraagd worden.
- *UI and rendering:* staat o.a. desktop notificaties toe.

Afhankelijk van welke categorie kan er een whitelist worden meegegeven die specifiek voor die categorie bepaalt welke operaties met welke parameters zijn toegestaan. Voor de categorie 'cookies' kan deze whitelist voor de leesoperatie bijvoorbeeld de lijst bevatten van de namen van de cookies die gelezen mogen worden. Standaard staat elke categorie op het niet toestaan van de bijhorende operaties. Figuur 3.1 geeft een voorbeeld van hoe een policy eruit zou kunnen zien. Policies worden volgens de JSON notatie [8] opgesteld. Op die manier kan de policy makkelijk omgezet worden tot een JavaScript object en kan er binnen het prototype makkelijk mee gewerkt worden.

```
{"cookies-read" : ["c_user"],
 "cookies-write" : ["c_user"],
  "geolocation" : "yes",
  "domaccess-read": "yes",
  "domaccess-write": "yes"
  }
```

Listing 3.1: Voorbeeld van een WebJail policy

Advice construction layer

De advice construction layer zet de hoog level policies van de policy layer om naar JavaScript adviesfuncties. Deze adviesfuncties zijn JavaScript functies die verantwoordelijk zijn voor het gecontroleerd uitvoeren van de JavaScript built-in functionaliteit. Zij zullen bepalen, op basis van de policies, of de functie mag uitgevoerd worden. De transformatie van policies naar adviesfuncties gebeurt in twee stappen. Voor elke categorie worden de bijhorende JavaScript functies geselecteerd. Vervolgens wordt voor elk van deze operaties een adviesfunctie gecreëerd op basis van de opgelegde policies.

Deep aspect weaving layer

Deze laatste layer is verantwoordelijk voor het vervangen van de originele implementatie door de adviesfunctie. Doordat dit de C++ operaties wijzigt in de browser worden alle toegangspaden naar de originele implementatie nu beveiligd door de adviesfunctie. De adviesfunctie is de enige plaats waar zich een referentie bevindt naar de built-in functionaliteit.

3.2 SES Prototype

In deze sectie stellen we de server-side versie van WebJail voor. De bedoeling is om de nood aan browserwijzigingen weg te werken en toch de vereiste veiligheid te garanderen. Deze eis kunnen we verzekeren door gebruik te maken van Secure EcmaScript [2]. We hebben in ons prototype revision 4779 van SES gebruikt. Het prototype richt zich specifiek op mashups met advertenties. We geven eerst een overzicht van de architectuur waarna we dieper ingaan op de implementatie.

3.2.1 Advertenties

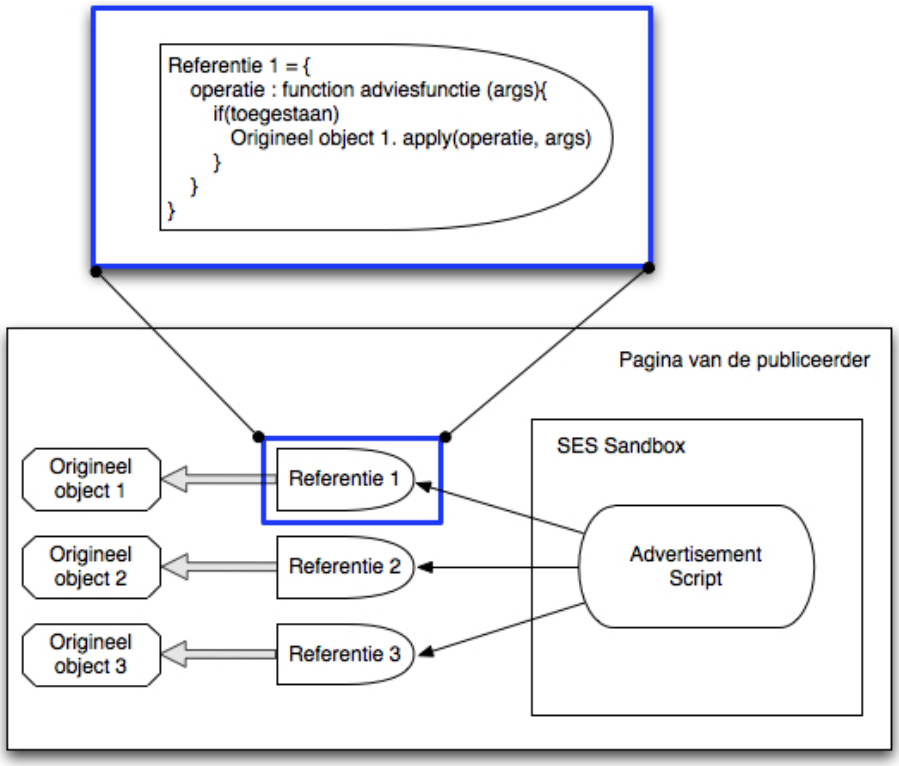
Doordat het prototype gericht is op het integreren van advertenties kunnen we enkele abstracties maken. De mashup integrator zullen we vanaf nu aanduiden als de publiceerder daar dit degene is die advertenties met behulp van het prototype zal opnemen in zijn pagina. Bij mashups op basis van advertenties is het de bedoeling het advertisment script dat verantwoordelijk is voor *targeting*, gecontroleerd uit te voeren in de pagina van de publiceerder. In het ontworpen prototype richten we ons dan ook vooral op de isolatie van zo een advertisement script. We kunnen ook enkele policy categorieën laten vallen. Advertisement scripts hebben bijvoorbeeld geen toegang nodig tot de data opslag interfaces (*Client-side storage*) van een web applicatie. Hetzelfde geldt voor de categorieën *UI and Rendering* en *Device Access*.

Bij het gebruik van een advertentienetwerk is het belangrijk dat de ad impressies correct gegenereerd worden. Dit zijn requests naar de server van de advertentienetwerken. Op basis van deze ad impressies wordt immers de vergoeding van de publiceerder berekend. Het is dus belangrijk dat het ontworpen SES prototype het aantal ad impressies niet beïnvloedt.

3.2.2 Architectuur

In grote lijnen komt het WebJail prototype gebaseerd op Secure EcmaScript overeen met de originele WebJail implementatie. Enkel de **Deep aspect weaving layer** is vervangen door de SES sandbox. Secure EcmaScript zorgt via een aantal beschikbare bibliotheken voor de creatie van een sandbox. Via referenties die gegenereerd zijn op basis van de aangereikte policies door de publiceerder, kan de JavaScript code binnen deze sandbox toegang krijgen tot de originele pagina.

In het SES prototype wordt het advertisement script geïsoleerd in deze SES sandbox. In de **Advice layer** worden de policies, opgelegd door de publiceerder, omgezet naar JavaScript functies, waarna deze beveiligde operaties via referenties beschikbaar worden gesteld aan het advertisement script in de sandbox. Deze referenties kunnen gezien worden als proxy-objecten die de operatie zal ontvangen van de sandbox. Wanneer toegestaan door de opgelegde policies, zal het referentieobject de oproep doorgeven aan het originele object. Figuur 3.3 geeft een visuele representatie van de architectuur van het SES prototype, toegepast op een advertisment script.



FIGUUR 3.3: Architectuur van het SES prototype

Policy layer

De policy layer is in grote mate hetzelfde gebleven als bij de originele implementatie. Net zoals bij de originele implementatie van WebJail kan de publiceerder policies opleggen aan het advertisement script. Door middel van een policy bestand dat ingeladen wordt, kan de publiceerder o.a. bepalen welke elementen hij toegankelijk wil maken.

In het oorspronkelijke concept van WebJail zijn een aantal policy categorieën aanwezig die niet van toepassing zijn op advertenties. *Client-side storage, UI and*

rendering en *device access* zijn niet nodig voor de ondersteuning van advertenties en worden dan ook niet toegankelijk gemaakt voor het advertisement script. Daar de SES sandbox isolatie als standaard neemt en dus initieel geen enkele referentie heeft naar buiten toe, hoeven hiervoor geen extra stappen ondernomen te worden. In het SES prototype wordt gecontroleerde toegang gegarandeerd voor de operaties die gerelateerd zijn aan DOM access, cookie access en het opvragen van de geolocatie. De categorie media wordt door het SES prototype niet ondersteund, dit omdat SES prototype een prototype is en de meeste advertenties dit niet nodig hebben. Een advertentie die gebruik maakt van een youtube filmpje zal dus niet werken. Dit ligt buiten de doelstellingen van deze masterproef.

Advice construction layer

Deze layer die oorspronkelijk verantwoordelijk is voor het omzetten van de policies naar JavaScript functies, heeft in het SES prototype de taak objecten te genereren die later zullen dienen als referenties. In deze referenties zitten de adviesfuncties impliciet vervat. Afhankelijk van de policies zal de referentie bepalen of de originele implementatie opgeroepen mag worden of niet.

Creatie van de Secure EcmaScript sandbox

De deep aspect weaving layer is vervangen door de creatie van de SES sandbox. Bij de creatie van deze afgeschermde module worden de referenties, gecreëerd door de advice construction layer, meegegeven. Op deze manier wordt het advertisement script geïsoleerd uitgevoerd en heeft het via deze referenties de nodige en gecontroleerde toegang tot de hoofdpagina. Er is geen nood aan wijzigingen in de browser waardoor er ook geen gebruikersondersteuning meer nodig is.

3.2.3 Implementatie

Policy layer

De policy wordt door de publiceerder opgesteld. Net zoals in de originele implementatie, wordt de policy volgens de JSON notatie opgesteld. Op die manier kunnen we deze makkelijk omzetten naar een JavaScript object zodat we hier later op een eenvoudige manier mee kunnen werken.

Advice construction layer

In het SES prototype van WebJail is de advice construction layer verantwoordelijk voor het opstellen van de referenties die later aan de SES sandbox zullen doorgegeven worden. Deze referentie vormt een soort van proxy-object dat eerst gaat controleren of de operatie mag uitgevoerd worden vooraleer de oproep doorgestuurd wordt naar het originele object. Impliciet zijn de adviesfuncties aanwezig in deze referenties.

DOM access

Voor de categorie DOM access gaat dit bijvoorbeeld als volgt in zijn werk: voor alle elementen op de hoofdpagina wordt een referentieobject aangemaakt dat aan de SES sandbox doorgegeven zal worden. Alle operaties die veiligheidsgevoelig zijn voor de DOM access worden toegevoegd aan dit object. We denken hier aan `appendChild`, `setAttribute`, `addEventListener` en nog een aantal andere cruciale JavaScript operaties die gelinkt zijn aan DOM access. De implementatie van de toegevoegde operatie gaat eerst kijken of de oproep geoorloofd is en pas wanneer dit het geval is, wordt de originele implementatie opgeroepen. De operaties van het referentieobject vormen dus de eigenlijke adviesfuncties. De SES sandbox zal op elk moment met zo een referentieobject werken en nooit rechtstreeks met het originele object. Op die manier is er altijd controle op de uitvoering van het advertisement script.

Listing 3.2 illustreert hoe dit bijvoorbeeld in zijn werk gaat voor `addEventListener`. *enforcedObject* stelt de referentie voor terwijl *originalEl* het originele element is.

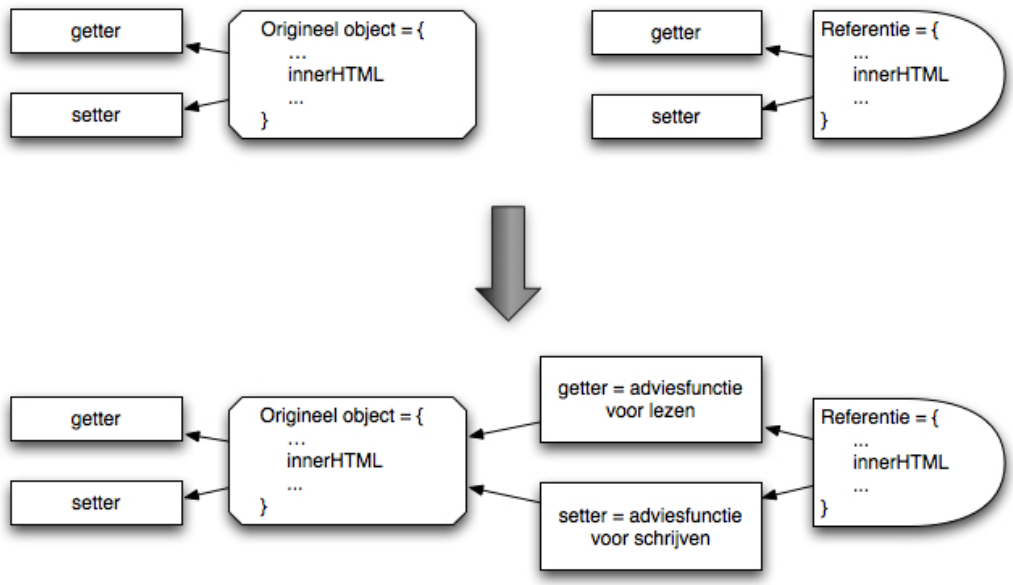
```
var write_access = (myPolicies["domaccess-write"] &&
    ((myPolicies["domaccess-write"] == "yes") ||
    (myPolicies["domaccess-write"].indexOf &&
    myPolicies["domaccess-write"].indexOf(element.id) != -1)))

enforcedObject.addEventListener = function(){
    if(write_access){
        return
        Element.prototype.addEventListener.apply(originalEl,arguments);
    }
    else{
        return;
    }
}
```

Listing 3.2: Adviesfunctie door addEventListener

Ook voor de attributen die te maken hebben met DOM access zoals `innerHTML`, `innerText`, `children`, etc. is een aanpak nodig die hetzelfde resultaat geeft. Omdat een attribuut zowel gelezen als geschreven kan worden, moeten attributen anders beveiligd worden. We konden deze niet zomaar vervangen door een functie omdat een attribuut gewoon een waarde voorstelt. Daarbij komt ook dat lezen en schrijven anders behandeld moeten worden afhankelijk van de policies. Om die reden hebben we dit op een alternatieve manier moeten verwezenlijken.

Elk referentieobject bevat ook de attributen van het originele object zoals `innerHTML`, `children`. Doordat attributen bij het wijzigen of lezen intern gebruik maken van *getters* en *setters*, kunnen de *getter* en *setter* voor een attribuut van het referentieobject vervangen worden door adviesfuncties. Deze adviesfuncties gaan eerst nakijken of het lezen en schrijven wel degelijk toegestaan is voor dat bepaald attribuut en geven dit, als het is toegestaan, door aan het attribuut van het originele object. Dit wordt visueel voorgesteld in figuur 3.4.



Figuur 3.4: Getter- en setterfunctie van de referentie vervangen door adviesfuncties.

```
enforcedObject.innerHTML = {};
MonitorPropertyEnforcedObject(element, enforcedObject, 'innerHTML',
    innerRead, innerWrite, read_access, write_access);

//Adviesfunctie
function innerRead(element, prop, read_access){
    var val;
    if(read_access){
        val = element[prop];
        return val;
    }
    val = "not readable";
    return val;
};
```

Listing 3.3: Voorbeeld van een adviesfunctie voor het `innerHTML`-attribuut

Listing 3.3 geeft een voorbeeld van hoe een adviesfunctie voor een attribuut werkt. Met behulp van de methode `MonitorPropertyEnforcedObject` worden de *setter* en de *getter* van de referentie (*enforcedObject*) vervangen door adviesfuncties (*innerRead* en *innerWrite*). Deze adviesfuncties zullen eerst controleren of de policy lees- en schrijftoegang toestaat. Indien de policy het toestaat, zal deze adviesfunctie de property van het originele object lezen of schrijven.

Geolocatie

De ondersteuning voor geolocatie is gelijkaardig aan die van DOM access. Voor geolocatie wordt een beveiligde referentie van `Navigator` doorgegeven aan de sandbox. In deze `Navigator` - referentie wordt dan weer een beveiligde referentie naar `geolocatie` opgeslagen die de nodige operaties bevat zoals bijvoorbeeld `getCurrentPosition()`, met daarin de controle van de policies. Op die manier kan bijvoorbeeld de huidige locatie door het advertisement script opgeroepen worden met *Navigator.geolocation.getCurrentPosition()*. De uitvoering hiervan zal onderhevig zijn aan de gedefinieerde policies.

Cookie

Het advertisement script kan de waarde van een cookie opvragen of wijzigen met behulp van `document.cookie`. De aanpak voor het beveiligen van het attribuut `cookie` werkt volledig analoog aan de controle van `innerHTML`. Aan de beveiligde referentie van `document` die zal doorgegeven worden aan de SES sandbox, wordt ook de eigenschap `cookie` toegevoegd. De *setter* en *getter* van het `cookie` attribuut in het referentieobject worden vervangen door adviesfuncties die wanneer de policies dit toestaan, de oproep doorgeven aan het originele object.

Script detectie

Om te voorkomen dat een script in de context van de hoofdpagina wordt uitgevoerd, is het belangrijk dat alle mogelijke manieren waarbij een script wordt toegevoegd aan de hoodpagina, beveiligd worden.

Een script kan op verschillende manieren aan een document worden toegevoegd. Een mogelijke manier om een tekstuele representatie van een script weg te schrijven is met behulp van de methode `document.write`. De originele implementatie van `document.write`, zij het gecontroleerd, beschikbaar stellen aan het advertisement script via een *document*-referentie, zou ervoor zorgen dat de hele pagina overschreven wordt bij een oproep aan deze methode. `document.write` is immer zo gedefinieerd dat het de hele pagina overschrijft wanneer de pagina al volledig geladen is op het tijdstip van de oproep. Ook werd het weggeschreven script uitgevoerd in de context van de hoofdpagina en niet in de sandbox waardoor het niet onderhevig was aan de opgelegde policies. Er moest dus een alternatieve manier gevonden worden om veilige maar analoge functionaliteit aan te bieden aan het advertisement script.

De `document.write` implementatie die beschikbaar gesteld wordt aan het advertisement script (via de *document* referentie), zal de weggeschreven tekst via het property `innerHTML` aan een specifiek div-element toevoegen. Alles wat het advertisement script wegschrijft met `document.write` zal aan deze div toegevoegd worden en vormt dus de plaats waar de advertentie terecht zal komen. (Dit is analoog aan de *AdJailDefaultZone* van AdJail [24]. Zie sectie 4.2.5 voor meer details.) Hierdoor is het probleem waarbij de gehele pagina overschreven wordt, opgelost. Scripts toegevoegd met `innerHTML` worden echter niet uitgevoerd terwijl dit wel het geval

was bij `document.write`. Om dit op te lossen kijken we welke script elementen recent toegevoegd zijn aan het eerder besproken div-element en voeren deze uit met *cajaVM.compileModule("script")(policy)*. Deze methode wordt beschikbaar gesteld door de SES bibliotheken en zorgt ervoor dat het script toch uitgevoerd wordt en dit binnen de geïsoleerde sandbox.

Een andere manier voor het toevoegen van een script aan de pagina, is via `appendChild`. Deze operatie voegt een element als kind toe aan een ander element. Zo kan een script dus aan het document gehecht worden door bijvoorbeeld *document.body.appendChild(script))*. Wanneer een script op die manier toegevoegd wordt, zou het script uitgevoerd worden in de context van de hoofdpagina.

Om die reden is het belangrijk dat alle objecten die uitgewisseld worden met de sandbox referentieobjecten zijn. Op die manier kan het geïsoleerde advertisement script op geen enkele manier wijzigingen uitvoeren die niet onderhevig zijn aan de opgelegde policies. In deze referenties zit immers altijd functionaliteit die zal nakijken of de operatie mag uitgevoerd worden volgens de opgelegde restricties.

Het script dat aangehecht wordt, zal immers aangemaakt zijn met behulp van de methode `document.createElement`. Deze methode zal een referentieobject van het aangemaakte script teruggeven. Wanneer er dan uiteindelijk JavaScript code wordt toegevoegd aan dit element, zal dit bijvoorbeeld gebeuren via het attribuut `innerHTML`. Doordat het advertisement script met een referentieobject werkt, wordt het zetten van `innerHTML` gecontroleerd en kunnen we, bij een script, de ingevoegde tekst opnieuw vervangen door *cajaVM.compileModule("originalInnerHTML")(policy)*. Wanneer het script dan via `appendChild` aan het document gehecht wordt, wordt dit toch uitgevoerd in de SES sandbox. Om een duidelijker beeld te vormen van hoe dit nu concreet in zijn werk gaat, geven we een praktisch voorbeeld.

Het script wordt aangemaakt met een gecontroleerde versie van createElement die beschikbaar gesteld is aan de sandbox. Deze gecontroleerde versie zal een referentieobject teruggeven van het aangemaakte script.

```
var scriptReferentie = document.createElement('script')
```

Omdat een referentieobject teruggegeven is, wordt het zetten van de innerHTML op dit object gecontroleerd.

```
scriptReferentie.innerHTML ="alert('test')"
```

Deze zal ervoor zorgen dat de originele tekst vervangen wordt door cajaVM.compileModule("alert('test')")
Wanneer het script dan met behulp van bijvoorbeeld `appendChild` wordt toegevoegd,
zal het script uitgevoerd worden binnen de SES sandbox.

Een andere vaak voorkomende manier voor het uitvoeren van scripts is het `src`-attribuut van een script-element zetten op de url van een extern JavaScript bestand. Deze externe code wordt dan uitgevoerd in de context van de pagina waaraan dit script is toegevoegd. Deze code moet dus net zoals andere scripts op één of andere manier uitgevoerd worden in de SES sandbox i.p.v. in de originele pagina.

Het idee was om de JavaScript code via een XMLHttpRequest (hierna afgekort als XHR) [4] op te halen van de gespecificeerde url en dan deze JavaScript code op de reeds besproken manier uit te voeren met behulp van *cajaVM.compileModule*. XHR is echter onderhevig aan het Same Origin Policy. Omdat de hoofdpagina zich in een ander domein bevindt als bron van de code kunnen we de code met behulp van XHR niet ophalen en dus ook niet uitvoeren in de SES sandbox. Op dit moment ondersteunen we om die reden nog geen scripts met een `src`-attribuut.

Event handling

Via het registeren van event handlers op HTML elementen is het ook mogelijk JavaScript uit te voeren. Wanneer een bepaald event plaats vindt, wordt automatisch de geregistreerde handler opgeroepen. Een handler kan op verschillende manieren geregistreerd worden. Met behulp van `addEventListener` of door een event attribuut van een element rechtstreeks te zetten bv. *element.onload = handler*. Wanneer de handler vanuit de sandbox geregistreerd wordt met `addEventListener` vormt dit geen probleem. De handler blijft immers binnen de sandbox en wordt ook binnen die context uitgevoerd wanneer het event optreedt. Bij het rechtstreeks zetten van een event attribuut geeft dit wel problemen. Eén van de vele manieren immers om een element via JavaScript toe te voegen aan de pagina, is een tekstuele representatie meegeven aan `innerHTML`. De handler, die via een event attribuut gelinkt wordt aan een element, wordt in deze situatie wel uitgevoerd binnen de context van de hoofdpagina. Hetzelfde gebeurt wanneer een tekstuele representatie via `document.write` wordt weggeschreven.

Om er voor te zorgen dat deze event handlers toch binnen de sandbox worden uitgevoerd, wordt bij het schrijven naar `innerHTML` elke event handler vervangen door *cajaVM.compileModule("handler")(policy)*. Dezelfde aanpak geldt voor `document.write` waar de event handlers uit de weg te schrijven tekst gefilterd worden.

Het opzoeken van de event attributen en hun handlers gebeurt met behulp van een reguliere expressie. Deze methode staat nog niet helemaal op punt omdat veel afhangt van het matchen van de juiste quotes maar voldoet voor het concept prototype. Een beter alternatief zou zijn te werken met een parser die de tekst omzet naar een structuur waaruit de attributen makkelijk te extracten zijn. Een volledig correcte parser vinden bleek echter niet zo éénvoudig. Hier kwam ook bij dat een parser de elementen omzet naar HTML elementen. Van zodra een HTML echter gecreëerd is, kan de handler uitgevoerd worden. Aangezien op dat moment de handler nog niet vervangen is door de sandbox expressie, zou het idee van isolatie volledig verloren

gaan.

Ad impressies

Aangezien elke operatie van het advertisement script slechts éénmaal wordt uitgevoerd binnen de sandbox, blijven de ad impressies correct. De uitvoering van het advertisement script wordt immers niet onderbroken of drastisch omgegooid. Het wordt enkel binnen een sandbox uitgevoerd en alle gegenereerde elementen waar request voor nodig zijn worden slechts éénmaal aangemaakt (wanneer toegestaan door de policies). Ook wanneer bepaalde operaties niet toegestaan zijn, worden deze niet uitgevoerd en worden de bijhorende ad impressies niet gegenereerd. Op die manier kan de publiceerder niet meer vergoed worden dan eigenlijk terecht is. Er hoeven dus geen extra stappen ondernomen te worden voor het behoud van deze ad impressies.

Als laatste dient nog opgemerkt te worden dat alle operaties en attributen van het originele object ook in de referenties opgenomen moeten worden, wil alle functionaliteit behouden blijven. Aangezien dit een enorm groot aantal is, hebben we in de context van een prototype enkel de belangrijkste operaties en attributen opgenomen en beveiligd zodat de werking van een advertentie gegarandeerd is.

Creatie van de Secure EcmaScript sandbox

De volgende stap is het aanmaken van de SES sandbox. De code van het advertisement script wordt opgehaald met XHR. Omdat XHR onderhevig is aan het same origin policy, kunnen we het advertisement script niet rechtstreeks van het advertentienetwerk ophalen. Het advertisement script moet dus aanwezig zijn op de eigen server aangezien hierdoor het same origin policy principe geen probleem meer vormt. Wanneer de code is opgehaald, wordt deze omgevormd tot een geïsoleerde module met behulp van de beschikbaar gestelde SES bibliotheken. Vervolgens worden de referenties toegankelijk gemaakt voor de module/sandbox. Listing 3.4 demonstreert de creatie van de sandbox.

```
var modMaker = cajaVM.compileModule(adSrc);
try {
    var returned = modMaker(references);
} catch(e) {
    console.log(e);
}

references = {
    document: enforcedDocument,
    window: enforcedWindow,
    navigator: enforcedNavigator
};
```

Listing 3.4: Creatie van de sandbox met een aantal mogelijke referenties.

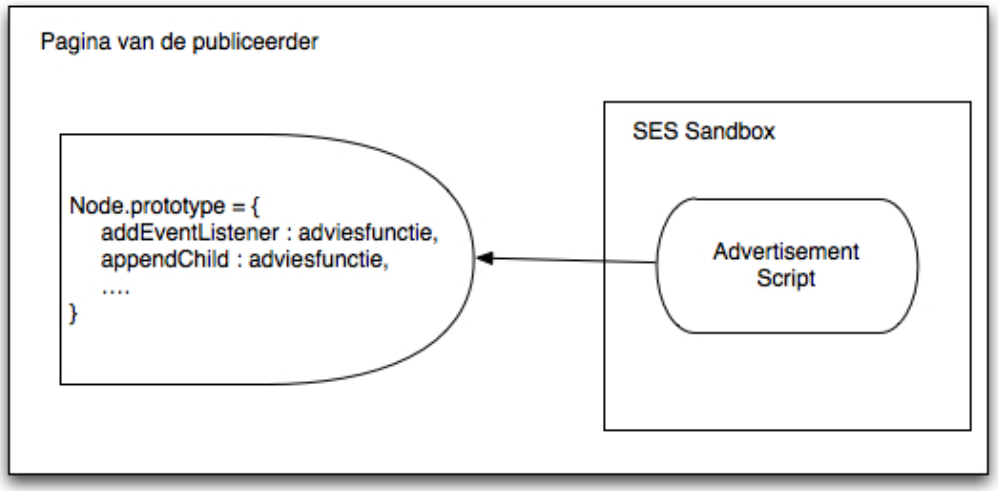
Vooraleer de publiceerder gebruik kan maken van het SES prototype, moet hij de SES bibliotheken importeren in zijn hoofdpagina zodat het prototype hiermee kan werken. Het volledige prototype zit vervat in de JavaScript bibliotheek *SES-prototype.js*. Deze bibliotheek biedt een operatie `loadAd` aan waaraan de url van het advertisement script en de url van de policies aan mee kunnen gegeven worden. Hierna wordt de SES sandbox opgesteld samen met de nodige referenties.

3.2.4 Overwogen alternatieven

Op dit moment worden voor alle HTML elementen waar de sandbox gebruik van maakt, referentieobjecten gecreëerd. Op die manier heeft het advertisement script de nodige functionaliteit en dit op een gecontroleerde manier. Voorafgaande aan deze implementatie, hadden we nog enkele alternatieven die elk om hun eigen reden niet toepasbaar waren.

Herdefiniëren van prototypes

In JavaScript gebeurt het overerven van methodes via het prototype. Elk element heeft dus een prototype waar het functionaliteit van overerft. Een overwogen alternatief was om de algemene implementatie van een methode te herdefiniëren in dit prototype zodat hierin de policies nagekeken konden worden, om vervolgens deze prototypes mee te geven aan de sandbox zodat deze prototypes gebruikt kunnen worden door de elementen in de sandbox. Visueel kan dit voorgesteld worden door figuur 3.5.



Figuur 3.5: Herdefiniëren van prototypes

Op die manier zouden alle elementen binnen de sandbox gebruik maken van de beveiligde operaties. Het herdefiniëren van prototypes binnen de sandbox op die

manier is echter niet mogelijk. Nieuw gecreëerde elementen binnen de SES sandbox bleven gebruik maken van de originele implementatie van het prototype, ook wanneer we een nieuw prototype meegaven voor hetzelfde type element. Deze manier van werken was dus geen optie.

Self-protecting JavaScript

Een tweede alternatief dat we overwogen hebben, was gebruik te maken van Self-protecting JavaScript [23]. Op die manier zouden we de originele implementatie kunnen beveiligen met behulp van de voorgestelde wrap-techniek. De originele implementatie zou dan vervangen worden door adviesfuncties dewelke kon garanderen dat de originele functionaliteit enkel opgeroepen werd wanneer toegestaan.

Het grote probleem hierbij was dat ook de functionaliteit van de hoofdpagina hiermee beveiligd werd, terwijl enkel de uitvoering van het advertisement script onderhevig moest zijn aan de opgelegde policies.

Hoofdstuk 4

AdJail Prototype

Op dit moment hebben we een WebJail prototype ontwikkeld dat gebaseerd is op Secure EcmaScript. Het SES prototype richt zich op mashups die advertenties integreren in de pagina van een publiceerder. Door de advertentie scripts te isoleren in een SES sandbox en hieraan enkel de referenties te geven die de advertentie nodig heeft, kan het prototype volgens het least-privilege principe controle afdwingen van de advertentie. Welke controle opgelegd moet worden, wordt bepaald door de opgestelde policies. Om de bruikbaarheid en de veiligheid van het ontwikkelde prototype na te gaan, hebben we een referentiepunt nodig zodat we kunnen vergelijken.

Daar AdJail, net zoals het eigen prototype, client-side beveiliging aanbiedt voor het integreren van advertenties in mashups, is deze geschikt om een vergelijkende analyse uit te voeren. In dit hoofdstuk zullen we dieper ingaan op onze implementatie van het AdJail concept, gebaseerd op de paper *AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements* [24].

4.1 Architectuur

Om de implementatie beter te kunnen plaatsen geven we nog een kort overzicht van de hoofdgedachte achter AdJail. Men wil vermijden dat advertisement scripts ongecontroleerd worden uitgevoerd in de context van de hoofdpagina waardoor o.a. alle informatie van de hoofdpagina toegankelijk is. AdJail biedt hiervoor een oplossing door het advertisement script te verplaatsen naar een schaduwpagina, geïsoleerd in een iframe. Door het origine van dit iframe verschillend te maken van de hoofdpagina (verschillend domein, protocol en poort), is het iframe onderhevig aan het same origin policy principe en kan het script dus niet aan vertrouwelijke informatie van de hoofdpagina.

Daar AdJail onder andere ontwikkeld is voor *contextual advertising* en dus informatie nodig heeft over de oorspronkelijke pagina, is het nodig dat de inhoud van de originele pagina wordt doorgestuurd naar de schaduwpagina. De publiceerder kan met behulp van policies zelf bepalen welke delen van zijn webpagina hij al dan niet

toegankelijk wil maken. Zo kan er bijvoorbeeld lees- en schrijftoegang toegekend worden aan een bepaald element op de pagina terwijl een ander element noch lees- noch schrijftoegang krijgt. Deze policies worden met behulp van een HTML attribuut gekoppeld aan de desbetreffende elementen en bepalen welke elementen uiteindelijk ook op de schaduwpagina zullen verschijnen.

Daar het script volledig geïsoleerd wordt uitgevoerd in de schaduwpagina zullen de wijzigingen van het script niet zichtbaar zijn op de hoofdpagina. Daarom is het nodig dat deze aanpassingen opgevangen worden en daarna worden doorgestuurd van het iframe naar de hoofdpagina. Hier zullen ze eerst worden gefilterd op basis van de opgelegde policies waarna ze al dan niet zullen toegepast worden op de hoofdpagina. Een impliciete policy die altijd geldt is dat scripts afkomstig van de schaduwpagina nooit gespiegeld worden naar de originele pagina. Hierdoor kan het advertisement script nooit code uitvoeren in de context van de hoofdpagina zonder onderhevig te zijn aan de andere policies. Dit hele synchronisatieproces tussen de schaduwpagina en de echte pagina heet **content mirroring**.

Ook de gebruikersreacties op de eventuele advertentie zoals bijvoorbeeld click-events op de originele pagina moeten worden doorgestuurd naar de schaduwpagina en worden daar door het advertisement script afgehandeld. Deze stap binnen het AdJail process heet **event forwarding**. Op deze manier is er op een gecontroleerde manier interactie tussen de gebruiker en het advertisement script.

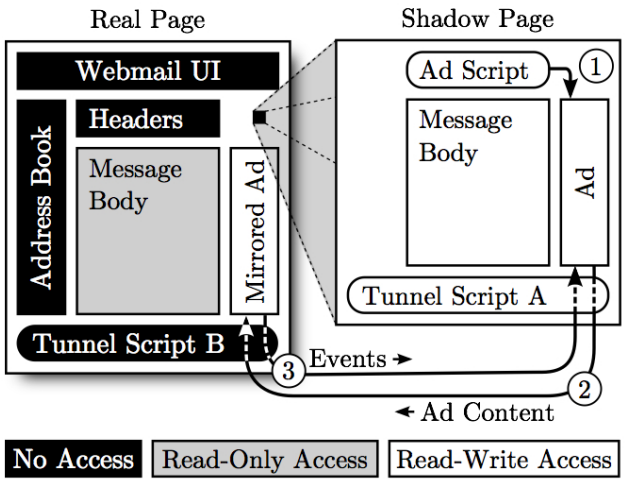
De communicatie tussen de hoofdpagina en de schaduwpagina gebeurt via inter-origin frame communicatie en is altijd onderhevig aan de eerder vermelde policies opgelegd door de publiceerder. Het uitwisselen van inhoud en het afdwingen van policies worden mogelijk gemaakt door tunnel scripts die toegevoegd worden aan zowel de hoofdpagina als de schaduwpagina. Hoe dit geheel juist samenwerkt, wordt geïllustreerd in figuur 4.1

4.2 Implementatie

In deze sectie gaan we gedetailleerd in op de eigen implementatie van AdJail. Eerst zal er besproken worden hoe de omhullende pagina en de schaduwpagina opgebouwd worden. Hier wordt ook een schets gemaakt van hoe de inhoud van de hoofdpagina naar de schaduwpagina verstuurd wordt. Vervolgens leggen we uit hoe de policies opgelegd worden en hoe event forwaring en content mirroring gerealiseerd worden en wat hier belangrijke aandachtspunten zijn. Als laatste vermelden we nog enkele problemen die we tijdens de ontwikkeling zijn tegengekomen.

4.2.1 Omhullende pagina versus schaduwpagina

Een belangrijke taak die door de publiceerder moet uitgevoerd worden vooraleer de bibliotheken van het AdJail prototype hun werk kunnen doen, is het toekennen



FIGUUR 4.1: Overzicht van AdJail toegepast op een webmail applicatie[24]

van policies aan de elementen die beveiligd moeten worden. In listing 4.1 wordt een template van de publiceerders pagina weergegeven.

```
<div id="MessageBody" policy="read-access:subtree;write-access:append">
    <img src="IMAGE_SRC" policy="read-access:subtree">
    <p style= "width: 1300px" policy="read-access:subtree">
    TEXT
    </p>
</div>

<script type="text/javascript" src="AdJail.js"></script>
<script>isolateAd('AD_URL','SHADOWPAGE_URL');</script>

<div id="AdJailDefaultZone" class="AdJailDefaultZone"
    policy="read-access:subtree;write-access:subtree;enable-iframe:allow">
</div>
```

LISTING 4.1: Template hoofdpagina

Een volgende belangrijke stap voor de implementatie van AdJail, is het isoleren van het advertisement script in een iframe. Dit iframe met de schaduwpagina wordt gecreëerd door het `src`-attribuut te laten verwijzen naar de locatie van de schaduwpagina. Deze schaduwpagina is een lege HTML pagina met enkel het schaduwtunnel script toegevoegd. Het is belangrijk dat deze schaduwpagina zich bevindt in een ander origine. Indien dit niet het geval is, is het Same Origin Policy concept niet van toepassing en gaat het hele idee van isolatie verloren. De publiceerder moet er dus voor zorgen dat de schaduwpagina tot een andere origine behoort.

Beter zou zijn om het iframe en de schaduwpagina volledig dynamisch aan te maken zodat de gebruiker zelf geen schaduwpagina in een ander domein moet voorzien. Echter omdat het iframe zich net bevindt in een ander domein, kunnen we niet aan de inhoud van de pagina waardoor het aanmaken van de schaduwpagina met de bijhorende functionaliteit dynamisch niet mogelijk is.

Het iframe met de schaduwpagina wordt onzichtbaar gemaakt zodat de gebruiker geen weet heeft van deze schaduwpagina. Hoe de creatie van dit iframe vertaald wordt naar JavaScript, wordt geïllusteerd in listing 4.2.

```
if (document.createElement && (iframe =
    document.createElement('iframe'))) {
        iframe.id = "shadowpage";
        iframe.name = "shadowpage";
        iframe.height = document.height;
        iframe.width = document.width;
        iframe.src = shadowpageUrl;
        iframe.style.display = "none";
        document.body.appendChild(iframe);
}
```

Listing 4.2: Creatie van de schaduwpagina geïsoleerd in een iframe

De AdJail bibliotheek is onder andere verantwoordelijk voor het doorsturen van de elementen naar de schaduwpagina, het afdwingen van de policies en de vereiste communicatie met de schaduwpagina voor onder andere event forwarding. Het biedt daarnaast een methode `isolateAd` aan waaraan de url van het advertisment script en de url van de schaduwpagina kan meegegeven worden. Op deze manier kan de publiceerder eenvoudig aanduiden welk script geïsoleerd moet worden en in welke schaduwpagina. Deze methode zal ervoor zorgen dat het advertisement script op het juiste moment (na het doorsturen van alle elementen) aan de schaduwpagina wordt toegevoegd zodat het geïsoleerd uitgevoerd wordt.

Vanwege contextual targeting wordt de schaduwpagina zelf gevuld met elementen afkomstig uit de hoofdpagina. Het tunnelscript zal de inhoud van de omhullende pagina scannen en modellen van aanwezige elementen doorsturen naar de schaduwpagina. Afhankelijk van welke policy opgelegd is op een element kan het model er anders gaan uitzien. Een element met enkel schrijftoegang en geen leestoegang zal bijvoorbeeld gemodelleerd worden als een leeg model. Een model met leestoegang daarentegen zal alle informatie over het desbetreffende element bevatten.

Bij het genereren wordt elk model eerst voorgesteld door een JavaScript object. Zo een object kan later eenvoudig omgezet worden naar een JSON representatie dewelke dan weer makkelijk kan doorgestuurd worden via inter-origin frame communicatie. Het model wordt opgebouwd door eerst enkele standaardeigenschappen (*nodeType, tagName, top, left, width,* en *height*) toe te voegen aan het model object. Vervolgens worden (bij leestoegang) alle overige attributen toegevoegd aan de attribuut-lijst.

Van de kindelementen wordt op zelfde manier een model opgebouwd, elk kindmodel wordt toegevoegd aan de rij met kinderen. Het opbouwen van een model gebeurt dus recursief.

Bij het genereren van deze modellen wordt voor elk element een ID gegenereerd. Op basis van dit ID worden de elementen op de hoofdpagina gelinkt aan de elementen op de schaduwpagina en kan dit later gebruikt worden voor synchronisatie. Een voorbeeld van hoe zo een model van een bepaald element er uiteindelijk uit zal zien, wordt geïllustreerd in listing 4.3.

```
{"nodeType":1,
 "tagName":"DIV","syncId":0,
 "top":8,"left":8,"width":1423,"height":52,
 "attributes":{
        "id":"MessageBody",
        "policy":"read-acces:subtree"
    },
 "children":[{"nodeType":3,"nodeValue":"\nHave you ever dreamed of
    owning your own car?}]
}
```

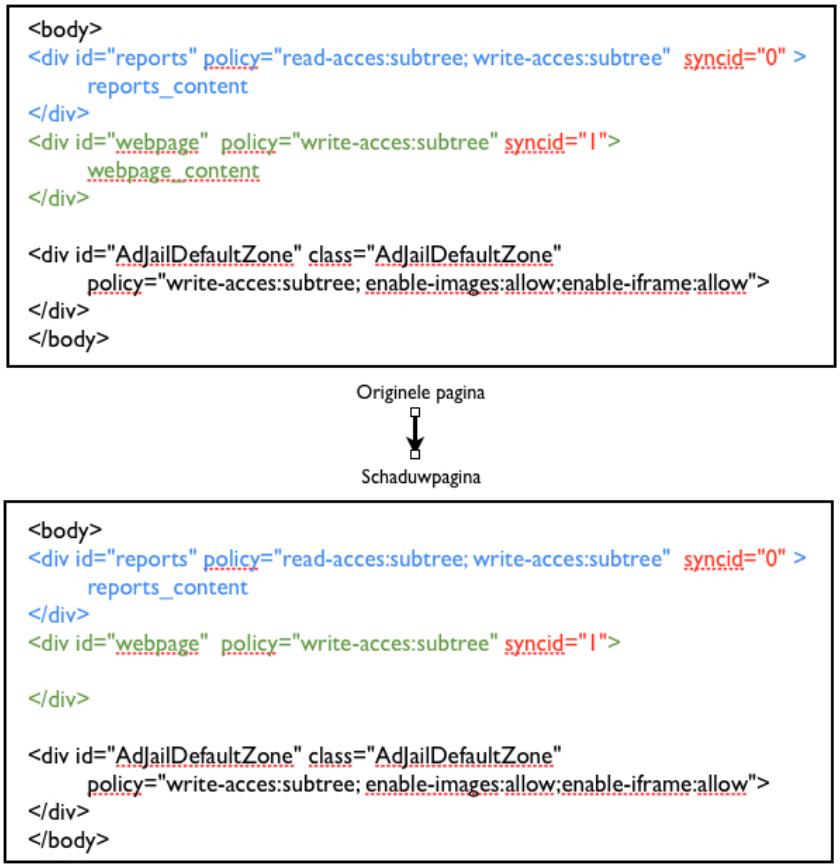
Listing 4.3: Voorbeeld van een model

Bij het scannen van de modellen wordt ook elk element toegevoegd aan een globaal toegankelijke lijst. Op deze manier kan bij het synchroniseren op basis van het synchronisatie ID het bijhorende element eenvoudig teruggevonden worden.

In zowel de originele pagina als de schaduwpagina moet een div-element aanwezig zijn dat de *AdJailDefaultZone* voorstelt en waar de eigenlijke advertentie uiteindelijke dient terecht te komen. Het advertisement script zal immers inhoud wegschrijven naar de pagina waarin het zich bevindt zonder dit aan een specifiek element toe te voegen. Door alles wat een advertisment script wegschrijft in deze zone terecht te laten komen en deze twee zones automatisch te synchroniseren, zullen we op die manier de advertentie ook te zien krijgen op de originele pagina. De *AdJailDefault-Zone* wordt aangemaakt door de publiceerder en samen met alle nodige elementen van de originele pagina naar de schaduwpagina doorgestuurd. Listing 4.2 toont een voorbeeld van hoe de originele pagina uiteindelijk geprojecteerd wordt op de bijhorende schaduwpagina.

4.2.2 Consistentie van de policies

Zoals eerder vermeld, kan elk HTML element geannoteerd worden met een policy. Een policy bestaat uit een lijst van permissies die elk een waarde kunnen hebben. In figuur 4.3 wordt een overzicht gegeven van de ondersteunde permissies binnen AdJail en hun bijhorende mogelijke waardes. Een policy kan er bijvoorbeeld als volgt uitzien : `policy:"write-access:subtree,enable-images:allow"`.



Figuur 4.2: Mapping van de originele pagina naar de schaduwpagina. (webpage-div wordt op een leeg element gemapt omdat er enkel naar mag geschreven worden)

Elke permissie staat default op de meest strenge waarde. Het is wel zo dat wanneer een ouder element een permissie heeft, het kind deze permissie automatisch ook heeft. Met behulp van het algoritme in figuur 4.4 wordt voor elk element aanwezig in de pagina een policy berekend. Dit algoritme werkt door de policy van het huidige element (indien die bestaat) te combineren met de policies van zijn ouder elementen en de meest restrictieve voor elke permissie over te houden. Verschillende policies aanwezig in een overervingsstructuur kunnen immers conflicteren. Een ouder element kan bijvoorbeeld write-access verbieden terwijl een kind-element write-access toestaat. Anderzijds kan een policy op zich tegenstrijdig zijn (bv. write-access: subtree en write-access: none in een zelfde policy). Deze inconsistenties worden door de methode `composePolicy` weggefilterd. Deze zal immers de nieuwe waarde voor een bepaalde permissie toevoegen aan de policy. Indien de permissie al aanwezig was in de bestaande policy, zal de meest restrictieve waarde overgehouden worden. Beide aangehaalde voorbeelden zullen hierdoor resulteren in een policy waarbij de permissie write-access op 'none' staat.

Permission	Values	Description / Effects
`read-access`	`none`†*, `subtree`	Controls read access to element's attributes and children.
`write-access`	`none`†*, `append`, `subtree`	Controls write access to element's attributes and children. Append is not inherited.
`enable-images`	`deny`†*, `allow`	Enables support in the whitelist for `<img>` elements, CSS `background-image` and CSS `list-style-image` properties.
`enable-iframe`	`deny`†*, `allow`	Enables `<iframe>` elements in whitelist.
`enable-flash`	`deny`†*, `allow`	Enables `<object>` elements of type `application/x-shockwave-flash` in whitelist.
`max-height`, `max-width`	`0`*, *n* `%`, *n* `cm`, *n* `em`, *n* `ex`, *n* `in`, *n* `mm`, *n* `pc`, *n* `pt`, *n* `px`, `none`†	Sets maximum height / width of element to *n* units. Smaller dimensions are more restrictive. When composing values specified in incompatible units, most ancestral value wins.
`overflow`	`deny`†*, `allow`	Content can overflow boundary of containing element if allowed.
`link-target`	`blank`*, `top`, `any`†	Force targets of `<a>` elements to `_blank` or `_top`. Not forced if set to `any`.

FIGUUR 4.3: Mogelijke permissies binnen AdJail

Algorithm 1: ComputePolicy(*targetElement*)

```
1  policy ← new Object();
2  WABeforeAppend ← undefined;
3  foreach element from root to targetElement do
4      if policy[ "write-access" ] = "append" then
5          policy[ "write-access" ] ← WABeforeAppend
6      statements ←Parse(
       element.getAttribute( "policy" ));
7      foreach stmt in statements do
8          policy ← ComposePolicies( policy,
           stmt );
9      if policy[ "write-access" ] ≠ "append" then
10         WABeforeAppend ← policy[ "write-access" ];
11 foreach permission in all permissions do
12     if permission is not defined in policy then
13         policy[ permission ] ← GetDefaultValue(
           permission );
14 return policy;
```

FIGUUR 4.4: Berekening van de policy voor een element aanwezig op de hoofdpagina. [24]

In onze implementatie hebben we er voor gekozen om voor elk element aanwezig op de pagina een policy te berekenen, dus ook wanneer er door de publiceerder geen policy is gedefinieerd voor een bepaald element. Op die manier kan de policy

onmiddellijk nagekeken worden wanneer het advertisement script dit element probeert te wijzigen. In het andere geval zou men bij één van de voorouders moeten gaan kijken welke policy deze heeft zodat deze kan toegepast worden. Ook wanneer er een nieuw element toegevoegd wordt door de advertentie, wordt er onmiddellijk een policy berekend voor het toegevoegde element. Op die manier kunnen we bij het berekenen ook onmiddellijk alle bestaande policies corrigeren wanneer ze inconsistent zijn.

Bij het berekenen van de policy zijn enkele uitzonderingen. Zo wordt de permissie `write-access:append` niet geërfd door de kindelementen. De append-policy betekent immers dat er wel kind elementen mogen toegevoegd worden aan het ouder element maar dat enkel deze toegevoegde elementen gewijzigd mogen worden. Bestaande kind elementen krijgen dus de write-permissie `none` toegekend. In tegenstelling tot de andere permissies krijgt de max-height permissie, de max-width permissie en de link-target permissie standaard de minst restrictieve waarde. Wanneer de publiceerder geen maximale hoogte of breedte oplegt, worden er geen beperkingen op dit vlak afgedwongen. Bij de link-target permissie wordt elke bestemming aanvaard.

4.2.3 Content mirroring

Een belangrijk onderdeel binnen AdJail is het synchroniseren van de omhullende pagina en de schaduwpagina. Wanneer het advertisement script een wijziging doorvoert aan de schaduwpagina, dient dit gespiegeld te worden naar de hoofdpagina. Het monitoren van de schaduwpagina kunnen we verwezenlijken door wrappers te plaatsen rond enkele cruciale JavaScript operaties. We denken hier aan de built-in functionaliteit `appendChild`, `replaceChild`, `removeChild` en `setAttribute` van het `Node` prototype, `setProperty` van het `CSSStyleDeclaration` prototype en `document.write`. Het wrappen van deze operaties vindt plaats in het tunnelscript op de schaduwpagina. Op die manier kunnen we wijzigingen afkomstig van het advertisement script eenvoudig onderscheppen. De aanpassing wordt gemodelleerd volgens het principe dat eerder in sectie 4.2.1 (JavaScript object naar JSON representatie) werd beschreven waarna dit model opnieuw via inter-origin communicatie wordt doorgestuurd naar de hoofdpagina.

Voor deze communicatie zijn enkel voorgedefinieerde boodschappen beschikbaar afhankelijk van welke operatie uitgevoerd moet worden. Elk bericht wordt opgebouwd als volgt: `operatie(parameter1 | parameter2 | ...)`. Tabel 4.1 geeft een overzicht van welke berichten van de schaduwpagina naar de hoofdpagina kunnen gestuurd worden.

Op de hoofdpagina wordt deze boodschap opgevangen. Daarna wordt de bijhorende operatie opgeroepen waarin wordt gecontroleerd of de wijziging voldoet aan de opgelegde policies. Wanneer het model niet voldoet aan de policies, wordt de aanpassing gewijzigd naar wel een aanvaardbare verandering (bijvoorbeeld de hoogte en de breedte van het toegevoegde element kunnen gewijzigd worden zodat de permissies

max-height en max-width niet meer geschonden worden.) Hiervoor hebben we een operatie `sanitizeModel` gedefinieerd die het model gaat updaten naar een versie die wel voldoet aan de opgelegde policy. Ook dit wordt recursief aangepakt, dus ook de modellen van de kinderen worden omgevormd naar een veilige versie. Wanneer dit niet mogelijk is, wordt de aanpassing eenvoudig geweigerd. Hieronder vallen bijvoorbeeld scripts daar deze onder geen enkele omstandigheden mogen toegevoegd worden aan de omhullende pagina. Doordat de controle van de policies uitgevoerd wordt door het tunnelscript op de hoofdpagina, kan het advertentie script onmogelijk invloed hebben op de policy controle.

`insertNode(syncId | model)` Voegt het meegegeven *model* toe als kind van het element geïdentificeerd door *syncId*.
`modifyAttribute(syncId | name | value)` Zet het attribuut *name* van het element geïdentificeerd door *syncId* op de meegegeven *value*.
`modifyStyle(syncId | name | value | priority)` Zet het CSS style attribuut *name* van het element geïdentificeerd door *syncId* op de meegegeven *value* en de meegegeven *priority*.
`modifyText(syncId | data)` Zet de tekst van het element geïdentificeerd door *syncId* op de meegegeven *data*.
`removeNode(syncId | childId)` Verwijdert het kind element geïdentificeerd door *childId* van het ouder element geïdentificeerd door *syncId*.
`replaceChild(syncIdParent | syncIdChild | model)` Vervangt het kind element geïdentificeerd door *syncIdChild* van het ouder element geïdentificeerd door *syncIdParent* door het meegegeven model.

TABEL 4.1: Overzicht van welke operaties naar de hoofdpagina kunnen gestuurd worden voor content mirroring.

In FireFox is het mogelijk om deze wrappers te verwijderen met de `delete` operator waarna de originele implementatie hersteld wordt. Omdat hierin echter geen policies gecontroleerd worden maar enkel de operatie doorgestuurd wordt naar de hoofdpagina, heeft dit geen enkele invloed op de veiligheid van AdJail.

Script detectie

Daar scripts onder geen enkele voorwaarde mogen uitgevoerd worden in de originele pagina, worden scripts niet gespiegeld. In de oorspronkelijke implementatie van AdJail wordt gebruik gemaakt van BluePrint [16] om scripts te detecteren. De aanpak van BluePrint elimineert de afhankelijkheid van de HTML parser aanwezig in de browser zelf. De standaardparsers van de browsers zijn te vergevingsgezind t.o.v. slecht gevormde HTML. Hierdoor kunnen scripts makkelijk geïnjecteerd worden

doordat ze niet herkend worden als scripts. BluePrint voorziet zelf een manier van parsen die ervoor zorgt dat er geen scripts aanwezig zijn in de resulterende boom van elementen. In onze implementatie van AdJail hebben we de techniek van BluePrint niet toegepast omdat dit ons te ver zou leiden van het doel van deze thesis. We hebben een naïeve aanpak geïmplementeerd die nakijkt of het woord 'script' voorkomt.

4.2.4 Event Handling

Bij vele advertenties is het zo dat listeners geregistreerd worden om de handelingen van de gebruikers op te vangen. Zo kan het zijn dat wanneer een gebruiker op een advertentie klikt, er een popup of dergelijke moet verschijnen. Omdat deze event handlers ook JavaScript functies zijn, mogen deze niet aan de hoofdpagina toegevoegd worden. Dit zou er immers voor zorgen dat eender welke JavaScript code uitgevoerd kan worden op de omhullende pagina. Daarom worden ook de operaties die verantwoordelijk zijn voor het registreren van handlers gewrapt. Het gaat hier over de JavaScript operaties `addEventListener` en `attachEvent` op het `Node` prototype. Wanneer er dan een listener geregistreerd wordt voor een bepaald event op een element, wordt dit door de wrapper doorgegeven aan de hoofdpagina. Op de hoofdpagina zal een voorgedefinieerde event handler geregistreerd worden in plaats van de handler die door het advertisement script werd aangereikt. De enige functie van deze voorgeregistreerde handler is het doorgeven aan de schaduwpagina van welk event op welk element heeft plaats gevonden. Hierbij wordt gebruik gemaakt van twee synchronisatieberichten die terug te vinden zijn in tabel 4.2.

`watchEvent(syncId | type | phase)` Bericht van schaduwpagina naar originele pagina. Registreert de zelf gedefinieerde event handler voor event *type* en *phase* op het element geïdentificeerd door *syncId*
`dispatchEvent(syncId | event)` Bericht van originele pagina naar schaduwpagina. Laat weten dat het event *event* op het element geïdentificeerd door `syncId` heeft plaats gevonden.

Tabel 4.2: Overzicht van welke operaties tussen de hoofdpagina en de schaduwpagina uitgewisseld kunnen worden voor event handling.

Listeners kunnen echter ook op andere manieren geregisteerd worden. Zo moet er rekening gehouden worden met het feit dat in JavaScript deze handlers ook als attribuut van een element kunnen gedefinieerd worden zoals in listing 4.4 het geval is. Een element kan op twee manieren aan de pagina toegevoegd worden nl. via de JavaScript operaties `appendChild` en `document.write`. In beide gevallen zal het element omgezet worden naar een model en verstuurd worden via het *insertNode* bericht uit tabel 4.1. Wanneer het aan de kant van de originele pagina opgevangen wordt, zal het, zoals vermeld in sectie 4.2.3, eerst door de methode `sanitizeModel` omgezet worden naar een veilige versie die voldoet aan de policies. Deze methode is

ook verantwoordelijk om de waarde van de attributen die een handler voorstellen te vervangen door de zelfgedefinieerde handler. Op die manier is het onmogelijk dat via het toekennen van een handler, code wordt toegevoegd aan de originele pagina.

```
<form><input id="button" type="button" name="test" value="Click me"
    onclick=alert("test")></form>
```

Listing 4.4: Voorbeeld van een listener meegegeven als attribuut

4.2.5 AdJailDefaultZone

Het advertisement script zal willekeurig dingen wegschrijven (met behulp van `document.write`) naar de pagina waarin het zich bevindt. Dit zal echter nooit worden getoond op de hoofdpagina waardoor de eigenlijke advertentie verloren zal gaan. Om deze reden, moet men gebruik maken van een *AdJailDefaultZone* die zowel op de hoofdpagina als op de schaduwpagina aanwezig is.

Op de schaduwpagina wordt het advertisement script in deze AdJailDefaultZone opgenomen. Alles wat het advertisement script dan wegschrijft naar deze AdJailDefaultZone, wordt automatisch doorgestuurd naar de AdJailDefaultZone op de hoofdpagina. Dit doen we door `document.write` te wrappen. Oorspronkelijk voegden we de tekst die weggeschreven werd met behulp van `document.write` toe aan de innerHTML van de AdJailDefaultZone en dit zowel aan de kant van de schaduwpagina als aan de kant van de originele pagina. (De oorspronkelijke functionaliteit van `document.write` overschrijft immers de hele pagina.) Het nadeel hiervan was dat we weinig controle konden uitoefenen op wat weggeschreven werd omdat onbekend was wat deze tekst inhield. Scripts konden nog weggefilterd worden door met behulp van een reguliere expressie na te gaan of het woord 'script' aanwezig was in de tekst maar nakijken of de breedte en hoogte niet overschreden werden, was moeilijk met deze aanpak.

Dit probleem hebben we kunnen oplossen door de weggeschreven tekst om te zetten naar de overeenkomstige HTML DOM elementen. Dit gebeurt door gebruik te maken van `innerHTML` van een hulp-div, vervolgens voegen we de kinderen van deze hulp-div één voor één als kindelement toe aan de AdJailDefaultZone. Het aanhechten van deze elementen gebeurt met behulp van `appendChild`. Doordat deze methode gewrapt is, zal elk element dat hiermee toegevoegd wordt, met behulp van een model doorgestuurd worden met het bericht `insertNode` naar de originele pagina waar het gecontroleerd zal worden op policies, eventHandlers etc.

Een probleem dat naar boven komt door het gebruik van `innerHTML`, is dat scripts die hiermee geparst zijn, niet uitgevoerd worden, zelfs niet wanneer ze achteraf aan de pagina worden toegevoegd met behulp van `appendChild`. Door bij elke oproep naar `document.write` na te kijken welke scripts toegevoegd zijn, kunnen we deze handmatig uitvoeren met `eval` binnen de context van de schaduwpagina en zo toch

de functionaliteit behouden.

Net zoals bij het SES prototype rijst hier nog het probleem met scripts-tags waarbij externe scripts worden ingeladen. Omdat deze door het gebruik van `innerHTML` niet worden uitgevoerd en er geen manier is om deze code handmatig uit te voeren, worden ook hier externe scripts niet ondersteund.

4.2.6 Ad Impressies

Zoals eerder vermeld zijn ad impressies requests die verstuurd worden naar de webserver van de adverteerder. Zo worden er onder andere ad impressies gegenereerd voor afbeeldingen, frames en objecten wanneer hun `src` attribuut gezet wordt. Daarnaast worden er ook request verstuurd voor CSS attributen zoals `background`, `background-image`, `list-style` en `list-style-image`. Omdat op basis van deze ad impressies de vergoeding van de publiceerder bepaald wordt, is het belangrijk dat deze correct gegenereerd worden. Doordat bij AdJail alles gespiegeld wordt op twee pagina's moet er een mechanisme zijn dat ervoor zorgt dat slechts alleen en alleen dan een request gestuurd wordt wanneer nodig. Dit wil zeggen enkel wanneer de advertentie getoond wordt op de originele pagina.

We kunnen hiervoor zorgen door de attributen van de elementen die ad impressies genereren, te vervangen door een plaatsvervanger. Daar een attribuut op verschillende manieren gezet kan worden, moeten al deze toegangspaden onderschept worden. In ons prototype hebben we enkel het behouden van ad impressies afkomstig van het `src` attribuut geïmplementeerd. Het correct genereren van impressies afkomstig van de andere attributen (`background-image`, etc.) kan op een analoge manier verwezenlijkt worden.

Een eerste manier is het zetten van het `src` attribuut via de operatie `setAttribute`. Deze werd sowieso al gewrapt omdat wijzigingen aan elementen moesten doorgestuurd worden naar de originele pagina. We hoeven in deze wrapper dus nog slechts extra functionaliteit toe te voegen. Wanneer de `setAttribute` operatie opgeroepen wordt met als parameter `src` wordt de `src` van het desbetreffende element niet gezet op de meegegeven parameter, maar ingevuld met een plaatsvervanger. Vervolgens zal met behulp van het bericht `modifyAttribute` wel de correcte parameter naar de originele pagina verstuurd worden. Op die manier bevat de schaduwpagina een plaatsvervanger terwijl de originele pagina toch de effectieve `src`-parameter bevat. Intern houden we ook een mapping bij van de synchronisatie id's van de elementen op hun originele waarde voor het `src`-attribuut. Zo kunnen we later de echte waarde eenvoudig terug bekomen. Dit is bijvoorbeeld nodig wanneer een element met een plaatsvervanger voor het `src`-veld als kind toegevoegd wordt aan een element met behulp van `appendChild`. Het model dat van dit kind doorgestuurd wordt naar de originele pagina moet het originele `src`-attribuut bevatten.

Daar de schaduwpagina nu een plaatsvervanger bevat, zal de originele implementatie van de operatie `getAttribute` indien opgeroepen op het `src`-attribuut, niet de originele `src`-waarde teruggeven. Daarom moeten we ook deze operatie wrappen zodat toch de correcte waarde wordt teruggegeven. Met behulp van de interne map uit vorige paragraaf kan de originele waarde probleemloos worden teruggevonden.

Een tweede manier om het `src`-attribuut te wijzigen is een eenvoudige toekenning zoals *element.src = "waarde"*. Het oorspronkelijke idee was om de setters en de getters van het `src`-attribuut voor alle elementen in één keer te herdefiniëren. Dit is echter niet mogelijk omdat de attributen van alle elementen dan dezelfde waarde zouden krijgen. Daarom onderscheppen we de operatie `document.createElement` die gebruikt wordt om HTML elementen aan te maken. We behouden de originele implementatie maar voegen een extra stap toe waarin we het `src`-attribuut voor elk object afzonderlijk gaan monitoren. Deze monitorstap zorgt ervoor dat de originele getter en setter vervangen wordt door zelfgedefinieerde methodes. In deze methodes wordt `getAttribute` en `setAttribute` respectievelijk opgeroepen. Deze werken zoals hierboven beschreven. Hoe dit specifiek voor het `src`-attribuut gebeurt, is terug te vinden in listing 4.5.

Een derde toegangspad dat onderschept moet worden, is het toevoegen van elementen via `document.write`. Het `src`-attribuut moet uit de weg-te-schrijven-tekst gehaald worden vooraleer het omgezet kan worden naar HTML DOM elementen. Van zodra het `src`-attribuut van een element immers gezet wordt, wordt onmiddellijk een request gegenereerd, ook al is het element nog niet toegevoegd aan de pagina. Het eerst parsen naar een DOM element en dan het `src`-veld vervangen door een plaatsvervanger is hier geen optie. De enige manier om dan de waarde van het `src` attribuut te vervangen, is met behulp van reguliere expressies. Daar we de originele waarde bijhouden in een interne map waarbij het synchronisatie id gebruikt wordt als sleutel, moeten we ook via deze reguliere expressies een synchronisatie id uit de tekst onttrekken. Wanneer er een id aanwezig is, wordt dit gebruikt als sleutel, anders wordt de naam van het element gebruikt.

Een laatste manier om het `src`-veld een waarde toe te kennen is via *element.innerHTML*. De browser heeft echter een interne definitie van innerHTML die moeilijk te bekomen is en daardoor moeilijk uit te breiden is. Het gewoon monitoren van deze property zoals het geval was bij een toekenning aan `src` kan hier dus niet toegepast worden.

Met behulp van proxies [21] kunnen we dit probleem oplossen. Een proxy vormt een object rond het originele object en geeft de operaties die opgeroepen worden (op deze proxy) door aan het originele object waarin nog steeds de originele implementatie aanwezig is. In de operaties van de proxy kunnen we echter eerst nog wat voorbereidend werk verrichten vooraleer we de operatie oproepen op het originele object. Door bij de implemenatie van `document.createElement` zo een proxy terug te geven kan men op die manier het lezen van of toekenningen aan *element.innerHTML* toch

nog gecontroleerd uitvoeren. Listing 4.5 geeft de uiteindelijke implementatie van `document.createElement`.

```
enforcePolicy({target:document,method:'createElement'},
    function(invocation){
        var element = invocation.proceed();
        if(element.tagName == 'IMG' || element.tagName == 'IFRAME' ||
            element.tagName == "OBJECT"){
            MonitorProperty(element,'src',srcRead,srcWrite);
            element = Proxy.create(handlerMaker(element));
        }
        return element;
})

function srcWrite(prop, oldval, newval){
    this.setAttribute("src",newval);
};

function srcRead(prop,newval){
    return this.getAttribute("src");;
};

function handlerMaker(obj) {
  return {
    ...
    get: function(receiver, name) { return
        replacePlaceHolderWithOriginalSrc(obj[name]); },
    set: function(receiver, name, val) { obj[name] =
        replaceSrcWithPlaceHolder(val); return true; },
    ...
  };
}
```

Listing 4.5: document.createElement geherdefinieerd.

Deze proxies worden echter op dit moment nog alleen maar ondersteund door Modzilla FireFox. Daar wij het AdJail prototype met de focus op Google Chrome hebben ontwikkeld, is dit niet ondersteund en ook niet aanwezig in onze implementatie.

4.2.7 Implementatie details

Een eerste probleem waar we mee in aanraking zijn gekomen is dat de modellen van de hoofdpagina al naar de schaduwpagina gestuurd werden vooraleer de listeners voor het ontvangen van berichten op de schaduwpagina geregistreerd waren. Hierdoor gingen de modellen verloren en werden deze ook niet toegevoegd aan de schaduwpagina. We hebben dit probleem verholpen door de schaduwpagina een bericht te laten sturen naar de hoofdpagina wanneer het klaar is voor het ontvangen van de modellen (en dus de listeners geregistreerd zijn).

Aansluitend hierbij was dat de modellen al gegenereerd werden voordat de afbeeldingen volledig geladen waren. Dit gaf als probleem dat de afmetingen nog niet

correct ingevuld waren (nul-waardes) waardoor de gespiegelde versies niet correct (foutieve afmetingen) werden toegevoegd aan de schaduwpagina. Door een event listener op het *onload* event van de hoofdpagina te plaatsen en dan pas het genereren van de schaduwpagina te starten, waren de afbeeldingen wel volledig geladen.

Hetzelfde probleem kwam aan bod bij de spiegeling van afbeeldingen naar de hoofdpagina. Wanneer een afbeelding door het advertisement script werd toegevoegd, moet een plaatsvervanger op de schaduwpagina geplaatst worden voor het behoud van de ad impressies zoals eerder al besproken werd. Het is echter belangrijk dat de afmetingen van de plaatsvervanger en de originele afbeelding overeenstemmen omdat het advertisement script anders incorrecte informatie krijgt over de locatie van zijn advertenties. De juiste positie is bijvoorbeeld belangrijk bij inline advertenties (advertenties die gelinkt worden aan woorden in een tekst).

Om de afmetingen te laten overeenstemmen, moeten de afmetingen van de originele afbeelding gekend zijn. Deze kunnen echter pas verkregen worden wanneer de afbeelding volledig geladen is, wat niet mag gebeuren aan de schaduwkant. We hebben dit opgelost door de afmetingen van de originele afbeelding, wanneer deze geladen is, terug te sturen naar de schaduwpagina en dan de afmetingen van de plaatsvervanger op basis hiervan aan te passen. Op die manier hebben de plaatsvervanger en de echte afbeelding dezelfde waarden voor hun afmetingen.

Hoofdstuk 5

Evaluatie: AdJail versus SES Prototype

In dit hoofdstuk komen we terug op zowel het prototype van AdJail als het SES prototype. Beide technologiën worden tegenover elkaar gesteld zodat de voordelen en nadelen/gebreken van beide technieken duidelijk naar voor komen. We hebben deze prototypes getest in Chrome 19.0.1084.53. We gaan eerst in op welk type policies door beide aanpakken ondersteund worden waarna we beide sandbox mechanismen tegenover elkaar stellen. Vervolgens kijken we naar de advertentienetwerken. In deze sectie geven we een overzicht van hoe advertentienetwerken werken, welke problemen op vlak van beveiliging dit met zich meebrengt en hoe deze eventueel in beide prototypes kunnen opgelost worden. Vervolgens worden de limitaties van beide manieren besproken alsook het toekomstig werk, om te eindigen met de doelstellingen die in het begin van deze thesis geformuleerd zijn.

5.1 Policies

Zowel AdJail als het SES prototype werken op basis van policies die door de publiceerder worden opgelegd. AdJail biedt echter vooral ondersteuning aan voor policies die te maken hebben met DOM access. Een element kan in AdJail bijvoorbeeld volgende policy opgelegd krijgen: '*policy = "write-access : subtree, read-access : subtree, enable-images : allow, max-height: 100px"*'. De AdJail architectuur zal er dan voor zorgen dat afbeeldingen of andere elementen die door het script toegevoegd worden aan het element met deze policy, conform zijn met deze policy. Aan operaties die niets te maken hebben met DOM access worden echter geen beperkingen opgelegd. Het advertisement script kan nog steeds aan het cookie, al is het binnen de context van het iframe. Ook het opvragen van de locatie is nog steeds mogelijk zonder enige controle. De focus bij AdJail ligt vooral op de controle van de inhoud die toegevoegd wordt aan en het lezen van de hoofdpagina.

Het SES prototype biedt wel ondersteuning voor het afdwingen van policies op operaties die niets te maken hebben met DOM access. Zo kan een policy verbieden

dat de geolocatie wordt opgevraagd of de toegang tot het cookie verbieden. In het SES prototype wordt dan weer weinig controle uitgevoerd op de inhoud die toegevoegd wordt aan de hoofdpagina. Een SES prototype policy kan bijvoorbeeld schrijftoegang toestaan op een bepaald element. Wanneer dit het geval is, kan vanaf dan elke inhoud naar dit element weggeschreven worden. AdJail controleert ook nog of afbeeldingen of frames worden toegestaan voor dit specifieke element en of hun afmetingen zoals bijvoorbeeld de maximale waarde niet overschreden is. De focus van het SES prototype ligt op de controle of een operatie mag uitgevoerd worden of niet.

Wanneer beide technieken tegenover elkaar worden gesteld, heeft het SES prototype meer mogelijkheden om de policy-ondersteuning uit te breiden. Bij AdJail worden alle operaties immers uitgevoerd in de schaduwpagina waar geen policies worden afgedwongen. Elke operatie mag dus uitgevoerd worden, zolang het in de schaduwpagina is. Enkel de doorgestuurde inhoud naar de hoofdpagina wordt gecontroleerd op basis van de policies. Een operatie zoals het opvragen van de huidige locatie kan dus moeilijk voorkomen worden aangezien deze zal uitgevoerd worden in de schaduwpagina.

Bij het SES prototype zijn er betere vooruitzichten. Aangezien elke operatie toch via een referentieobject langs een eventuele adviesfunctie met policies moet vooraleer de originele functionaliteit uitgevoerd wordt, kan de toegevoegde inhoud daar gecontroleerd worden.

5.2 Sandbox mechanismen

Beide manieren zijn gebaseerd op hetzelfde idee. Ze isoleren JavaScript code in een sandbox zodat ze de code gecontroleerd kunnen uitvoeren. Bij AdJail wordt die sandbox voorgesteld door een iframe dat zich bevindt in een andere origine. Hierdoor is de context van de hoofdpagina afgeschermd van de code in dit iframe. Bij het SES prototype wordt de sandbox aangeboden door Secure EcmaScript. Door de code te transformeren naar een veilige subset wordt de code een security capability language, met andere woorden een sandbox die enkel toegang heeft tot de omhullende pagina via expliciet toegekende referenties.

Beide manieren van werken hebben voor- en nadelen. Aangezien bij AdJail alles uitgevoerd wordt in de schaduwpagina, is het belangrijk dat alles correct gespiegeld wordt. Dit brengt extra complexiteit met zich mee aangezien bijvoorbeeld de reeds besproken ad impressies behouden moeten worden. In het SES prototype wordt de code enkel uitgevoerd wanneer het effectief toegestaan is door de policies. De wijzigingen afkomstig van het advertisement script worden dan ook onmiddellijk aangebracht op de hoofdpagina. Dus niet zoals bij AdJail waar het script sowieso uitgevoerd wordt en alleen de toegestane wijzigingen gespiegeld worden. Het behoud van ad impressies is bij het SES prototype dus impliciet aanwezig.

5.3 Advertentienetwerken

In deze sectie geven we een beschrijving van hoe verschillende advertentie scripts afkomstig van advertentienetwerken in hun werk gaan. Vervolgens geven we aan hoe deze door beide prototypes ondersteund worden en welke problemen dit met zich meebrengt.

5.3.1 Werking

De meeste advertisement scripts voegen advertenties toe d.m.v. iframes. Deze iframes worden met behulp van `document.write` toegevoegd aan de hoofdpagina. Binnen deze manier van werken zijn er nog 2 variaties.

Een eerste manier die gebruikt wordt door advertisement scripts is het iframe in zijn geheel wegschrijven. Het `src`-attribuut is onmiddellijk ingevuld met de link naar een pagina die in het iframe zal ingeladen worden. Op die manier zit het iframe ook in een andere origine en is het onderhevig aan het same origin policy principe. De inhoud van het iframe is volledig afgeschermd van de hoofdpagina. Een advertentienetwerk dat op deze manier te werk gaat is o.a. Bidvertiser [6].

Een tweede aanpak is een leeg iframe wegschrijven naar de hoofdpagina en pas daarna met de `document.write` van het iframe-document (dus niet de `document.write` van de hoofdpagina!) de advertentie toevoegen aan dit iframe. In dit geval is het belangrijk op te merken dat het toegevoegde iframe in hetzelfde domein zit als de hoofdpagina. Het same origin policy principe biedt hier dus geen bescherming. De inhoud die toegevoegd wordt aan deze iframes zijn vaak script-tags die op hun beurt dan weer een iframe gaan toevoegen aan het huidige iframe. Dit nieuwe iframe werkt analoog aan de eerst besproken manier en is dus wel onderhevig aan het same origin policy principe. Advertentienetwerken die deze aanpak gebruiken zijn o.a. Google AdSense [14] en AdBrite [5].

Een belangrijke waarneming bij advertisement scripts afkomstig van deze advertentienetwerken is dat geen enkel netwerk gebruik maakt van DOM access. Ondanks dat zowel Google Adsense, AdBrite en Bidvertiser contextual advertising aanbieden. We hebben met Google AdSense de test gedaan. Wanneer een advertentie van Google Adsense geplaatst wordt op een pagina, is deze advertentie initieel niet aangepast aan de pagina-inhoud. Na enkele dagen bleek dat de advertentie uiteindelijk toch toegespitst was op het onderwerp van de pagina. Dit komt doordat Google AdSense gebruik maakt van een 'AdSense-crawler' die de webpagina bezoekt om de inhoud ervan te bepalen. Pas wanneer deze crawler de pagina verwerkt heeft, kunnen relevante advertenties getoond worden volgens de gecachte versie [13].

In het geval van de andere twee advertentienetwerken, moet de publiceerder vooraf met enkele kernwoorden aangeven waarover zijn webpagina zal handelen. Dit gebeurt bij de inschrijvingsprocedure die de publiceerder moet voltooien, wil hij

advertenties van zulke netwerken opnemen in zijn pagina. Op basis hiervan wordt een link gegenereerd die de publiceerder met behulp van de HTML script tag kan toevoegen aan zijn webpagina.

5.3.2 Ondersteuning

Dat advertisement scripts veelvuldig gebruik maken van iframes levert enkele problemen op voor zowel het AdJail prototype als het SES prototype.

Bij de eerste aanpak waarbij het iframe volledig wordt weggeschreven, werkt AdJail zoals het hoort (indien iframes uiteraard zijn toegestaan door de publiceerder). Natuurlijk, wanneer de publiceerder geen afbeeldingen toestaan maar wel iframes, kan niet gecontroleerd worden of aan de eis van geen afbeeldingen voldaan is. De inhoud van het iframe is niet onderhevig aan de opgelegde polcies. Wegens het same origin policy principe kunnen we immers niet aan de inhoud van dit iframe.

In het SES prototype levert zo een iframe echter andere problemen. Scripts die binnen dit iframe worden uitgevoerd staan volledig los van de SES sandbox. Ze worden binnen de context van het iframe uitgevoerd en kunnen dus aan alle operaties. Het opvragen van de geolocatie bijvoorbeeld is opnieuw mogelijk. Ook wanneer de policies dit niet toestaan. Omdat in dit geval het iframe behoort tot een andere origine kunnen we hier weinig aan veranderen. Wegens de bescherming die aangeboden wordt door het same origin policy principe kunnen we, zoals eerder vermeld, immers niet aan de inhoud van het iframe en dus ook niet aan de JavaScript code die uitgevoerd wordt binnen het iframe. Er is dus geen enkele manier om rond deze JavaScript code een SES sandbox te plaatsen.

In het tweede geval waarbij eerst een leeg iframe wordt weggeschreven en pas daarna de inhoud wordt toegevoegd, liggen de problemen anders. AdJail werkt in dit geval niet zoals het hoort. Het iframe wordt wel toegevoegd maar de inhoud die achteraf weggeschreven wordt, wordt niet gespiegeld naar de hoofdpagina. Dit komt omdat de inhoud aan dit iframe toegevoegd wordt met behulp van de `document.write` van dit iframe zelf. In het door ons geïmplementeerde prototype wordt echter enkel een wrapper geplaatst rond de `document.write` van de schaduwpagina die niet dezelfde operatie is als die van het iframe. Op die manier wordt de inhoud van het iframe niet gespiegeld.

In het geval van het SES prototype kunnen we wel in beperkte mate voorkomen dat operaties ongeoorloofd uitgevoerd worden. In dit geval valt het iframe wel onder hetzelfde origine als de hoofdpagina. Aangezien het toevoegen van inhoud aan iframes altijd via referentieobjecten gebeurt, kunnen we nog enkele aanpassingen aan dit iframe toevoegen voordat de scripts worden ingeladen. Zo kunnen we bij het aanmaken van het iframe de SES bibliotheken toevoegen en zo de sandbox beschikbaar maken. Ook alles wat weggeschreven zal worden naar dit iframe zal

gebeuren via referentieobjecten. Bij het wegschrijven van scripts naar dit iframe kunnen we de operaties dan opnieuw vervangen door de expressie die al meerdere malen aan bod is gekomen nl. *CajaVM.compileModule('script')(policy)* die er voor zal zorgen dat de JavaScript code toch binnen de sandbox uitgevoerd wordt. Dus de code in listing 5.1 wordt omgezet in de code van listing 5.2.

```
iframe.contentDocument.write("<script>window.alert('should be executed
    in sandbox')</script>");
```

LISTING 5.1: Script wegschrijven in een iframe.

```
iframe.contentDocument.write(
"<script>cajaVM.compileModule('window.alert(" +
    '"' + "should be executed in sandbox" + '"' +
    ")')(policy)</script>"
);
```

LISTING 5.2: Script wegschrijven in een iframe dat uitgevoerd zal worden in de sandbox. (De vele quotes zijn nodig voor de correcte uitvoering van de string.)

Uiteindelijk zal er echter toch opnieuw een iframe van een andere origine weggeschreven worden. Zoals eerder vermeld kan hier weinig aan verholpen worden.

Advertisement scripts schrijven ook vaak hun eigen scripts weg in kleine stukjes, o.a. AdBrite werkt op deze manier. Hoe dit werkt, wordt geïllustreerd in listing 5.3.

```
C.write("<scr");
C.write('ipt src="script.js"></scr');
C.write("ipt>\n");
```

LISTING 5.3: Een script opgesplitst wegschrijven.

Bij AdJail geeft dit echter functionele problemen. De implementatie van `document.write` wordt immers aan de schaduwkant op een alternatieve manier geïmplementeerd. Zoals beschreven in sectie 4.2.5, wordt de weggeschreven tekst eerst geparst met behulp van `innerHTML` waarna het aan de AdJailDefaultZone toegevoegd wordt. `innerHTML` aanvaardt helaas geen halve tags. Hierdoor wordt '<scr' niet toegevoegd aan de AdJailDefaultZone. Het voorbeeld in listing 5.3 zal uiteindelijk resulteren in de tekst *'ipt src="script.js">ipt>'*. Dit zal er natuurlijk voor zorgen dat het script niet wordt uitgevoerd waaruit volgt dat het advertisement script van AdBrite niet correct werkt in combinatie met AdJail. Naast het functionele probleem komt ook dat een opgesplitst script moeilijk te detecteren is aangezien deze deeltjes afzonderlijk weinig betekenis hebben.

Ook bij het SES prototype wordt de weggeschreven inhoud aan een specifiek div toegevoegd met behulp van `innerHTML`. Hierbij komen dus dezelfde problemen bovendrijven.

Een oplossing voor dit probleem werkt door opeenvolgende oproepen naar `document.write` te bufferen tot een volledige tag ontvangen is. Op die manier zal `innerHTML` de volledige tag ontvangen waardoor er geen delen worden geweigerd. Hierdoor kan een script ook makkelijker gedetecteerd worden en door het desbetreffende prototype correct afgehandeld worden.

5.4 Veiligheid

Om de veiligheid te evalueren, hebben we geprobeerd om externe code uit te voeren buiten de sandbox van beide prototypes. We hebben ons gebaseerd op het XSS Cheat Sheet [15] en daar aanvallen uit de verschillende categoriën embedded scripts, event handler en URI attributen, getest.

Embedded script elementen

Een eerste manier om een script te injecteren, is om een script te 'verstoppen' in een ander element. Een voorbeeld van een scriptinjectievector ziet er uit als volgt:

```
<IMG "" "><SCRIPT>window.alert("XSS")</SCRIPT>">
```

Wanneer deze tekstuele voorstelling met behulp van `document.write` wordt weggeschreven, wimpelen beide prototypes deze aanval af. Dit komt omdat in beide prototypes de implementatie van `document.write` verwezenlijkt wordt door de weg-te-schrijven tekst toe te voegen aan `innerHTML`. De implementatie van `innerHTML` werkt echter zo dat scripts niet worden uitgevoerd wanneer ze hieraan toegevoegd worden. Om er toch voor te zorgen dat er geen gebrek aan functionaliteit optreedt, moesten we een manier vinden om het toegevoegde script toch uit te voeren.

In het geval van het SES prototype wordt gekeken naar welke scripts recent zijn toegevoegd. Vervolgens worden ze met behulp van de expressie *cajaVM.compileModule(script)(policy)* uitgevoerd binnen de sandbox. Het idee van embedded scripts is om een script er niet te laten uitzien als een script. Ook al slaagt een aanvaller in dit opzet, de code zal niet uitgevoerd worden aangezien een uitvoering enkel verwezenlijkt kan worden met een expliciete oproep naar *cajaVM.compileModule(script)(policy)*.

Het AdJail prototype werkt volledig analoog. Aangezien geïnjecteerde code uitgevoerd mag worden aan de schaduwkant, worden de toegevoegde scripts uitgevoerd met `eval`, altijd in de context van de schaduwpagina. Bij AdJail blijft natuurlijk nog het gevaar dat het script via een model doorgestuurd wordt naar de hoofdpagina. Aan de hoofdpagina is een mechanisme aanwezig dat het model veilig zal maken vooraleer de inhoud wordt toegevoegd aan de pagina. In deze stap zullen scripts dus geëlimineerd worden. In ons AdJail prototype is dit een naïeve implementatie met reguliere expressies maar zoals eerder vermeld maakt de originele implementatie gebruik van BluePrint. Deze aanpak biedt weerstand tegen verschillende injectievectoren.

Event handler

Een manier voor het injecteren van code is via event handlers. Beide prototypes bieden hier oplossingen voor. In het prototype van SES worden handlers binnen de SES sandbox uitgevoerd terwijl event handlers in AdJail steeds binnen de schaduwpagina blijven. Voor verdere details verwijzen we naar de sectie 3.2.3 voor de implementatie van het SES prototype en sectie 4.2 voor de implementatie van het AdJail prototype.

URI attributen

Een manier die aangehaald wordt in de XSS Cheat Sheet is het injecteren van scripts via het `src`-attribuut. Dit gaat bijvoorbeeld in zijn werk als volgt:

```
<IMG SRC="javascript:alert('XSS')"
```

Deze injectievector wordt door de browser zelf niet meer ondersteund waardoor dit ook geen gevaar meer vormt voor de prototypes van zowel AdJail als SES.

5.5 Limitaties

Een belangrijke tekortkoming aan het SES prototype is dat scripts waaraan een JavaScript bestand wordt meegegeven via het `src`-attribuut, niet ondersteund worden. De scripts toegevoegd met `innerHTML` worden zoals eerder besproken niet uitgevoerd. Bij scripts waarvan de code expliciet aanwezig is in het 'body' van het script element, kunnen we dit handmatig uitvoeren met *cajaVM.compileModule*. Aan deze expressie moet de code echter als tekst meegegeven worden. In het geval waarbij een extern JavaScript bestand wordt gebruikt in het script, is de code als tekst niet beschikbaar omdat XHR gelimiteerd is door het same origin policy principe. Hierdoor kunnen we code dus niet opvragen om deze in de SES sandbox uit te voeren. Dit zal resulteren in een gebrek aan functionaliteit wanneer een advertisement script deze scripts wil toevoegen aan de pagina. Omdat AdJail op een analoge manier werkt (scripts toegevoegd met document.write worden uiteindelijk ook via `innerHTML` aan de pagina toegevoegd), komen hier dezelfde problemen aan bod.

Bij de geteste netwerken geeft dit geen problemen voor AdSense en Bidvertiser. Deze advertentienetwerken voegen enkel scripts met een src toe aan hun iframe. Zoals eerder besproken worden deze niet binnen de sandbox uitgevoerd waardoor dit geen problemen geeft. Bij de voorgestelde oplossing voor iframes van hetzelfde origine zou dit wel een probleem geven (zie sectie 5.3.2). AdBrite gaat anders te werk. Deze laatste schrijft een script weg waar een extern JavaScript bestand wordt ingeladen. Dit externe JavaScript bestand zal de uiteindelijke advertentie toevoegen. Aangezien dit extern bestand niet zal uitgevoerd worden, werkt AdBrite niet correct met beide prototypes.

Een andere limitatie van het SES prototype vloeit voort uit het gebruik van Secure EcmaScript. Om de code tot een veilige versie te kunnen transformeren, moet aan een aantal voorwaarden voldaan zijn. Zo moeten variabelen o.a. altijd voorafgegaan worden door `var`. Wanneer dit niet het geval is, zal de code niet omgevormd kunnen worden en dus ook niet gebruikt kunnen worden tenzij de publiceerder wijzigingen gaat doorvoeren aan het advertisement script. Bidvertiser gaf bijvoorbeeld problemen op dit vlak aangezien het op regelmatige basis gebruik maakte van variabelen die niet met `var` gedeclareerd waren.

Het feit dat de code getransformeerd moet worden, brengt een volgende limitatie met zich mee. Om de code te kunnen omvormen, is er nog steeds een dichte koppeling met de code. Om te kunnen werken met de SES sandbox moet de publiceerder immers in het bezit zijn van de code. Zoals eerder vermeld is het dynamisch ophalen van de JavaScript code onderhevig aan het same origin policy principe. Enkel wanneer de advertentienetwerken specifiek het domein van de publiceerder zouden toelaten om XHR request uit te voeren, zou het dynamisch kunnen verlopen.

Als laatste blijft ook het probleem van iframes bestaan. Elk advertentienetwerk zal uiteindelijk een iframe wegschrijven dat zich bevindt in een ander domein. Binnen dit iframe kunnen alle operaties nog steeds uitgevoerd worden, al is het binnen de context van dit iframe. Advertentienetwerken werken op deze manier zodat ze enkele veiligheidsgaranties kunnen aanbieden. De advertentienetwerken verzekeren zo dat de advertenties van de willekeurige adverteerders niet aan de hoofdpagina kunnen. Enkel het script afkomstig van het advertentienetwerk heeft toegang tot de hoofdpagina. Een mogelijkheid zou zijn om iframes te weigeren, maar ook dit zal resulteren in een gebrek aan functionaliteit aangezien deze iframes gebruikt worden om advertenties in te laden.

5.6 Toekomstig werk

Binnen het SES prototype zijn nog enkele delen vatbaar voor verbetering. De implementatie maakt gebruik van reguliere expressies om event handlers uit weggeschreven-tekst te filteren. Binnen het concept 'prototype' voldeed deze manier van werken. De gebruikte reguliere expressie is echter gevoelig aan de verschillende quotes die binnen een textuele representatie kunnen gebruikt worden nl. de enkele quote en de dubbele quotes. Een betere manier om dit probleem op te lossen, is het gebruik van een parser die event handlers onmiddellijk vervangt door een veilige implementatie.

Ook bij AdJail wordt gewerkt met een reguliere expressie om het `src`-attribuut te vervangen door een plaatsvervanger. Dezelfde argumentatie als bij het SES prototype geldt wat betreft deze reguliere expressie.

Op het huidige moment is er voor het SES prototype slechts een conceptueel idee wat betreft de veiligheid rond iframes. Ook voor scripts met een `src`-attribuut moet nog een oplossing gevonden worden hoe aan de code te komen zodat deze in de sandbox kan opgenomen worden. De implementatie moet hiermee dus nog uitgebreid worden.

5.7 De doelstellingen hernomen

Tot slot hernemen we nog eens de vooropgesteld doelstellingen:

Doelstelling 1 De client mag niet gewijzigd worden. WebJail wordt vanuit de server gepusht naar de client. De client hoeft zelf geen actie te ondernemen. Het is niet de bedoeling dat het afdwingen van de policies zelf gecontroleerd wordt aan de server kant. Dit zal nog steeds client-side gebeuren maar zonder extra ondersteuning van de gebruiker.

Met behulp van Secure EcmaScript zijn we er in geslaagd deze doelstelling te bereiken. Er is geen nood meer aan het wijzigen van de browser.

Doelstelling 2 De mashup integrator kan op elke component een policy opleggen die omschrijft wat de component wel en niet mag. Concreet wil dit zeggen dat de policy vastlegt tot welke operaties de component toegang heeft.

De publiceerder kan met behulp van een policy bestand policies opleggen aan het advertisement script. Deze policy zal bepalen welke operaties via referentieobjecten beschikbaar worden gemaakt voor het advertisement script. Ook deze doelstelling is bereikt.

Doelstelling 3 De mashup integrator heeft de verantwoordelijkheid voor het afdwingen van de policies op de componenten. De component moet dus zelf geen specifieke ondersteuning aanbieden om deel te kunnen uitmaken van de mashup (bijvoorbeeld beperkte JavaScript set).

Om Secure EcmaScript te kunnen gebruiken moet de JavaScript code correct gevormd zijn. Zo moeten variabelen altijd voorafgegaan worden door `var` en is het gebruik van `caller` en `callee` verboden. Wanneer hier niet aan voldaan is, kan de code niet omgevormd worden tot een veilige versie. Andere ondersteuning van de component is niet nodig. Aan deze doelstelling is dus niet volledig voldaan.

Doelstelling 4 De attacker code die eventueel in de component aanwezig is, kan de beveiliging niet omzeilen. De built-in functies kunnen niet uitgevoerd worden zonder dat de desbetreffende policies er op worden afgedwongen.

Vele advertisement scripts maken gebruik van iframes. Binnen deze iframes worden geen policies opgelegd. Omdat deze iframes zich vaak bevinden in een andere origine en dus volledig afgeschermd zijn door het same origin policy principe, is het ook niet altijd mogelijk om een SES sandbox in te laden. Enkel wanneer de JavaScript code binnen de SES sandbox wordt uitgevoerd, is aan deze doelstelling voldaan.

Hoofdstuk 6

Besluit

We zijn er in geslaagd om een eigen WebJail prototype gebaseerd op Secure ECMAScript (SES prototype) te ontwikkelen waardoor advertisement scripts gecontroleerd en in isolatie uitgevoerd kunnen worden. Het doel om de gebruikersondersteuning weg te werken uit WebJail is bereikt. Door het gebruik van Secure ECMAScript zijn wijzigingen in de browser niet langer nodig.

De toegang tot de pagina van de publiceerder gebeurt gecontroleerd waardoor contextual advertising mogelijk blijft zonder dat de publiceerder gevaar loopt. Met behulp van policies kan hij immers zelf bepalen tot wat het script toegang heeft. Ook dit was een voorop gestelde doelstelling die volbracht is. Naast deze doelstellingen was het ook belangrijk dat de aanbieder van advertenties zelfs geen ondersteuning moet bieden om opgenomen te kunnen worden in de pagina van de publiceerder. Door het gebruik van Secure ECMAScript moet de JavaScript van advertisement script echter voldoen aan enkele strikte regels. Deze doelstelling is dus jammer genoeg niet bereikt.

Bij de vergelijking met AdJail kwam naar boven dat het ontwikkelde SES prototype dezelfde beveiliging aanbiedt en zelfs meer. AdJail beschermt vooral de data van de hoofdpagina en dwingt controle af op de inhoud die door het advertisement script wordt toegevoegd. Het prototype gebaseerd op Secure ECMAScript biedt ook controle aan voor andere operaties. We denken hier bijvoorbeeld aan het opvragen van de geolocatie waarvoor geen data van de hoofdpagina nodig is. Het verbieden van deze operatie is binnen AdJail niet mogelijk. Het SES prototype richt zich meer op de controle of een operatie mag opgeroepen worden of niet, terwijl AdJail zich meer richt op de inhoud die toegevoegd wordt. Wanneer een advertisement script bijvoorbeeld een afbeelding wil toevoegen aan de hoofdpagina, zal AdJail zich meer bekommeren over het feit dat de afbeelding voldoet aan bv. bepaalde afmetingen, en of afbeeldingen wel toegestaan zijn door de pagina. Het SES prototype zal zich in dit geval enkel focussen of er wel elementen mogen toegevoegd worden zonder zich vragen te stellen over wat toegevoegd wordt. Wel moet opgemerkt worden dat het SES prototype de mogelijkheid biedt uitgebreid te worden zodat het het controleren van de toegevoegde inhoud toch ondersteunt.

We kwamen echter ook tot de vaststelling dat de meeste advertentienetwerken geen gebruik maken van gecontroleerde toegang tot de hoofdpagina maar dat ze hun advertenties op een andere manier afstemmen op de pagina-inhoud. Hierdoor gaat een groot deel van het nut van beide prototypes verloren. Beide prototypes hebben immers deels als doel deze toegang veilig te laten plaatsvinden. Het nut gaat natuurlijk niet helemaal verloren. Bij AdJail kan de publiceerder nog steeds bepalen aan welke voorwaarde de inhoud, die toegevoegd wordt door de advertenties, moet voldoen. Het SES prototype controleert naast toegang tot de hoofdpagina ook nog andere operaties zoals het opvragen van de geolocatie.

Het SES prototype heeft wel nog enkele belangrijke limitaties. Zo kan een extern ingeladen script niet uitgevoerd worden tenzij de publiceerder dit script in zijn bezit heeft. Dit is een groot nadeel aangezien deze manier van werken, waarbij externe scripts worden ingeladen, vaak gebruikt wordt. Het bemachtigen van deze externe scripts is niet altijd vanzelfsprekend. Dit probleem komt ook bij AdJail naar voor. Net zoals bij het SES prototype, worden extern ingeladen scripts niet ondersteund door AdJail.

Naast dit probleem kan de beveiliging die het SES prototype aanbiedt, omzeild worden met behulp van JavaScript iframes. Deze worden gebruikt om externe webpagina's op te nemen in een andere webpagina. Binnen deze iframes worden scripts immers niet binnen de SES sandbox uitgevoerd. Wanneer het domein van dit iframe overeenstemt met het domein van de pagina van de publiceerder, kan hier nog wel een oplossing voor gevonden worden. In dit geval kunnen we de inhoud van het iframe nog wijzigen en het SES isolatiemechanisme binnen dit iframe initialiseren alvorens scripts in dit iframe worden uitgevoerd. Wanneer dit iframe echter behoort tot een ander domein, verbiedt de browser alle toegang tot dit iframe. In dit geval kunnen we weinig veranderen aan de ongecontroleerde uitvoering van scripts binnen dit iframe. De laatste doelstelling stelde dat er geen enkele mogelijkheid mocht zijn om de policies te omzeilen. Aangezien dit toch mogelijk is met iframes, is deze doelstelling niet bereikt.

Aangezien advertentienetwerken veelvuldig gebruik maken van deze iframes en externe scripts, werkt de voorlopige versie van zowel AdJail als het SES prototype helaas niet optimaal in combinatie met deze advertentienetwerken.

Bijlagen

Bijlage A

Populariserend artikel

Facebook zonder zorgen
Vang hackers in een JavaScript sandbox

Lynsey Hens & Brenda Lissens

3 juni 2012

1 Inleiding

Dynamische web pagina's en mashups zijn door het huidige internet tijdperk niet meer weg te denken uit onze dagelijkse bezigheden. Internet gebruikers maken meermaals gebruik van zulke mashups zonder zich bewust te zijn van de gevaren die deze met zich meebrengen. Op dit moment zijn er al een reeks veiligheidstoepassingen bekent die hier tegen bescherming bieden, elk met hun eigen voor- en nadelen.

Een mashup is een applicatie die bestaande toepassingen samenbrengt, eventueel zelf gedefiniëerde functionaliteit toevoegt en hierdoor een meerwaarde biedt aan de gebruiker. Meer specifiek zal dit artikel dieper ingaan op web mashups uitgevoerd in een browser. Hieronder vallen webpagina's die andere online diensten in hun pagina opnemen en op die manier nieuwe toepassingen beschikbaar maken voor online gebruikers. iGoogle is een populair voorbeeld dat duidelijk het begrip van web mashups illustreert. De gebruiker kan op éénvoudige wijze bestaande web applicaties toevoegen aan iGoogle zoals Facebook, Twitter, weerberichten, etc en biedt zo een overzicht aan de gebruiker. Zulke interactieve programma's kunnen met behulp van verschillende technieken verwezenlijkt worden. Hierbij moet echter rekening gehouden worden met de voorwaarden waaraan een mashup moet voldoen. Zo kan het voor verschillende componenten wenselijk zijn dat deze onderling kunnen samenwerken (functionaliteit) terwijl tegelijkertijd verwacht wordt dat de geïntegreerde delen elkaar niet ongewenst gaan verstoren (veiligheid).

De ideale oplossing die de nodige veiligheidsmaatregelen en de gevraagde functionaliteit aanbiedt, wordt samengevat door het least-privilege principe. Dit principe beschrijft dat elke component slechts de rechten mag hebben die hij nodig heeft voor zijn kernfunctionaliteit. Op deze manier verliest de mashup noch aan functionaliteit noch aan veiligheid. De meest voorkomende technieken die momenteel gebruikt worden om componenten te integreren in een mashup zijn script inclusie en iframe integratie dewelke niet gebaseerd zijn op het least-privilege principe. Deze manieren

van implementatie brengen dan ook een aantal veiligheidsproblemen met zich mee.

Bij script inclusie wordt de code in dezelfde context als de omhullende pagina uitgevoerd waardoor de componenten toegang krijgen tot de data van deze omhullende pagina. Hier worden dus geen veiligheidsmaatregelen aan de componenten opgelegd. De component heeft dezelfde rechten als de omhullende pagina en kan dus ook aan alle informatie. Deze data kunnen interessante informatie bevatten voor buitenstaanders. De geschiedenis van de webbrowser of de cookies kunnen bijvoorbeeld een waardevolle betekenis hebben voor externe partijen. Hiernaast kan de ene component van de mashup de andere component kwaadwillig beïnvloeden en ook dit kan tot gevaarlijke gevolgen leiden.

Hier tegenover staat iframe integratie. De component lijkt in een aparte pagina geladen te zijn en heeft dus zijn eigen omgeving waardoor de component onderhevig is aan het Same Origin Policy principe. Het Same Origin Policy, een veiligheidsmaatregel afgedwongen door de browser, legt vast dat scripts van verschillende webpagina's (verschillend domein, protocol of poortnummer) elkaars methodes en waarden niet kunnen gebruiken. Op deze manier wordt verhinderd dat een component aan de gevoelige informatie van de mashup pagina kan. Dit belet echter niet dat de bijhorende krachtige operaties toch nog uitgevoerd kunnen worden, al is het enkel gebruik makend van informatie binnen de context van de component.

In volgende paragrafen gaan we dieper in op Caja en Self-Protecting JavaScript, twee technieken die script inclusie uitbreiden om zo toch het least-privileged principe te bekomen.

2 Caja

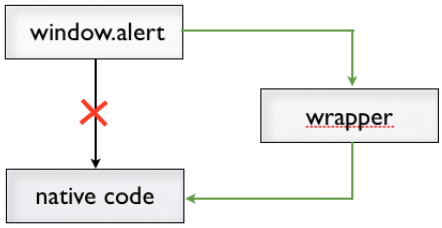
Een voorbeeld van een techniek die verder bouwt op script inclusie is Caja [2] [5]. Caja [2] [5] realiseert het least-privilege principe door de code van de geïmporteerde component om te zetten naar een veilige subset. Deze veilige subset stelt een capability security language voor. Het principe achter een capability security language houdt in dat een object alleen maar toegang heeft tot een ander object enkel en alleen als het een referentie heeft naar dat object. Standaard heeft een object geen referentie. Wanneer een object moet communiceren met een ander object worden er referenties toegevoegd. Op deze manier kan men een sterke controle houden over de invloeden/capabilities van een object en kan men het least-privileged principe makkelijk verwezenlijken aangezien rechten in de vorm van referenties kunnen toegekend worden. Deze veilige JavaScript subset wordt gemaakt door de externe code eerste te transformeren (cajolen) en dan pas in de mashup op te nemen.

Caja[2] brengt echter ook een aantal nadelen met zich mee. Zo moet de code van een component altijd eerst gecajoled worden voor het gebruikt mag worden in

de mashup. Hierdoor ontstaat er een heel dichte koppeling tussen de beveiliging en de code van de geïntegreerde component. Er worden ook beperkingen opgelegd op het gebruik van bepaalde JavaScript methoden en objecten. Het gebruik van eval, with, caller, callee, __defineGetter__, ... is bijvoorbeeld niet toegelaten. Wanneer dit wel gebruikt wordt in de op te nemen component, kan deze niet gecajoled worden en dus niet gebruikt worden met Caja [2], wat een beperking aan functionaliteit met zich meebrengt.

3 Self-Protecting JavaScript

Self-protecting JavaScript [6] [7] [3] is een techniek die, net zoals Caja, gebaseerd is op script inclusie. Bij deze techniek wordt er in de header van de webpagina een extra JavaScript bestand toegevoegd die wrappers plaatsen rond de native JavaScript functies. Op deze manier kunnen er policies opgelegd worden op built-in operaties en kan de uitvoering ervan gecontroleerd worden. De originele functie wordt dus vervangen door een JavaScript wrapper die als enige een referentie bevat naar de werkelijke built-in. In deze wrapper zit de policy vervat en enkel wanneer er aan de policy voldaan is, zal de wrapper toelaten de oorspronkelijke functionaliteit op te roepen. De wrap-functionaliteit wordt in de header van de web pagina toegevoegd. Hierdoor is het wrappen van de te beveiligen functies het eerste wat op de web pagina uitgevoerd wordt en is het niet mogelijk dat externe code nog voor het wrappen, een referentie bemachtigt naar de originele implementatie.

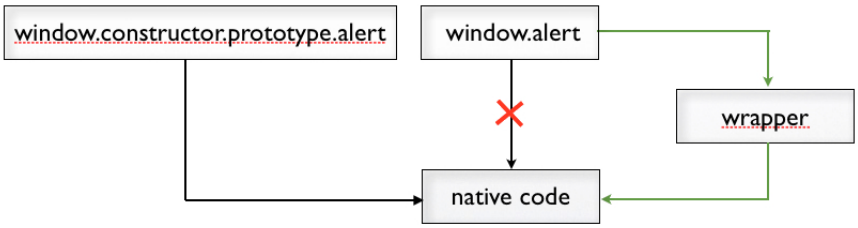


Figuur 1: Voorbeeld van een wrapper

Self-Protecting JavaScript [6] [7] [3] heeft als voordeel dat het een lichtgewicht aanpak is. Het is namelijk niet nodig om code te gaan transformeren en worden er ook geen beperkingen opgelegd aan het gebruik van JavaScript. Deze manier van werken biedt dus de mogelijkheid aan om policies op te leggen aan built-in functies. Daar dit wrappen echter op een hoog niveau aangeboden wordt, blijven er enkele belangrijke nadelen aanwezig.

Eén van de belangrijkste nadelen van Self-Protecting JavaScript is dat de wrapper slechts één alias naar de originele functie wrapt. Zo kan de native functie 'alert'

rechtstreeks aangeroepen worden op het window-object maar kan het ook uitgevoerd worden door langs de constructor en het prototype te gaan. Wanneer echter window.alert gewrapt wordt, blijft het pad via de constructor en het prototype onbeveiligd.



Figuur 2: Voorbeeld waarbij de wrapper omzeilt wordt

Andere zwakheden waar Self-protecting JavaScript [3] aan lijdt, zijn het herdefiniëren van methodes die gebruikt worden in de policy. Een aanvaller kan bijvoorbeeld de apply functie zo herschrijven dat de methode waarop deze functie wordt opgeroepen, opgeslagen wordt in een variabele die voor de aanvaller toegankelijk is. Zo kan hij toch de beschermde functionaliteit ongeoorloofd uitvoeren. Op deze aanval bestaat ook een variant waarbij men de waarde van het caller en callee argument uitbuit. Een specifiek voorbeeld van deze aanval is het overschrijven van de toString-implementatie. Wanneer de originele alert functie wordt aangeroepen, zal deze de toString methode aanroepen de welke dan aan de originele implementatie van alert kan via het caller en callee argument.

Een volgend gekend probleem is het aanmaken van nieuwe windows en frames. Vanuit deze nieuw aangemaakte omgevingen kan de aanvaller opnieuw de originele implementatie van built-in functionaliteit herstellen.

Het valt niet te ontkennen dat Self-Protecting JavaScript [6] [7] [3] nog lijdt aan enkele belangrijke zwakheden. Zwakheden die om voldoende veiligheid te kunnen bieden, weg gewerkt moeten worden.

Zowel Caja als Self-Protecting JavaScript, voldoen niet aan de voorwaarden van mashups. Deze mashups dienen maximale beveiliging te combineren met optimale functionaliteit zoals het geval zou zijn bij het least-privilege principe. Een alternatieve techniek is WebJail [1], een toepassing van het least-privilege principe die verder bouwt op iframe integratie.

4 WebJail

De architectuur van WebJail [1] is er in geslaagd om het least-privilege principe succesvol toe te passen op mashups. Zoals eerder vermeldt, bouwt WebJail [1] verder op iframe integratie. Hierdoor maakt WebJail [1] gebruik van de voordelen van het Same Origin Policy principe waardoor de geïntegreerde componenten geen toegang hebben tot de data van de omhullende pagina. Deze beveiligingsmaatregel is echter niet voldoende aangezien vele operaties nog steeds ongecontroleerd kunnen toegepast worden. WebJail [1] tracht hier een oplossing voor te bieden.

De mashup kan aan elke component waarvan hij functionaliteit importeert, een policy opleggen. Deze policy bepaalt voor elke component afzonderlijk tot welke categorie van operaties de desbetreffende component al dan niet toegang heeft. Zo kan een policy toestaan dat de component gebruik maakt van de mediafaciliteiten van de integrerende pagina maar toegang tot de geolocatie van het systeem verbieden. Deze policies die gedefinïeerd zijn voor categorieën van operaties, worden omgezet naar advies functies voor elke JavaScript operatie afzonderlijk. Bij het oproepen van één van deze gevoelige operaties, zal deze advies functie bepalen of de operatie mag uitgevoerd worden of niet. WebJail [1] maakt gebruik van *Deep Aspect Weaving*. Dit wil zeggen dat door wijzigingen in de browser het op geen enkele manier mogelijk is, de originele implementatie van een operatie te bemachtigen zonder eerst door de advies functie te gaan. Hierdoor zal een component er nooit in slagen, ongeoorloofd een krachtige operatie uit te oefenen.

5 Probleemstelling

WebJail [1] slaagt er in om het least-privilege principe toe te passen maar maakt gebruik van het principe van Deep Aspect Weaving dat eerder ook bij ConScript [4] werd gebruikt. Dit heeft als voordeel dat het onmogelijk is de advies functie te omzeilen maar hier is ook een belangrijk nadeel aan verbonden. Om te voorkomen dat een beschermde operatie langs geen enkele andere weg bekomen kan worden, moeten er wijzigingen aangebracht worden in de browser. De C++ operaties aanwezig in de browser worden zo gewijzigd dat de advies functie eerst wordt uitgevoerd. Hiervoor is ondersteuning van de browser en dus impliciet van de gebruiker nodig. Wanneer de implementatie van WebJail [1] wijzigt, zal de gebruiker hier actief moeten op reageren door eventueel updates te installeren. De huidige client-side implementatie van WebJail [1] vervangen door een server-side uitwerking is de belangrijkste doelstelling van deze thesis waarbij het least-privileged principe behouden blijft. Dit doel gecombineerd met de oorspronkelijke implementatie van WebJail [1] resulteert in de volgende requirements:

- **Requirement 1** De client mag niet gewijzigd worden. WebJail [1] wordt vanuit de server gepusht naar de client. De client hoeft zelf geen actie te

ondernemen. Het is niet de bedoeling dat het afdwingen van de policies zelf, gecontroleerd wordt aan de server kant. Dit zal nog steeds client-side gebeuren maar zonder extra ondersteuning van de gebruiker.

- **Requirement 2** De mashup integrator kan op elke component een policy opleggen die omschrijft wat de component wel en niet mag.
- **Requirement 3** De mashup integrator draagt de verantwoordelijkheid voor het afdwingen van de policies op de componenten. De component moet dus zelf geen specifieke ondersteuning aanbieden om deel te kunnen uitmaken van de mashup. (bijvoorbeeld beperkte JavaScript set)
- **Requirement 4** De attacker code die eventueel in de component aanwezig is, kan de beveiliging niet omzeilen. De built-in functies kunnen niet uitgevoerd worden zonder dat de desbetreffende policies er op worden afgedwongen.

Naar alle waarschijnlijkheid zullen deze requirements verwezenlijkt kunnen worden door gebruik te maken van (onderdelen van) de bestaande technieken Caja [2] [5] en Self-Protecting JavaScript [6] [7] [3].

6 Conclusie

Het gebruik van web mashups brengt een aantal veiligheidsrisico's met zich mee. Het least-priviliged principe dat ervoor zorg dat een component slechts die rechten krijgt die het nodig heeft voor zijn kernfunctionaliteit, wordt niet toegepast door meest gebruikte technieken op dit moment, nl. iframe integratie en script inclusie. Caja en Self-Protecting JavaScript zijn technieken die verderbouwen op script inclusion om het least-priviliged principe te verwezenlijken. Beide technieken brengen echter een aantal nadelen met zich mee. Hier tegenover staat WebJail, een techniek die verderbouwt op iframe integratie maar waarvoor wijzigingen in de browser nodig zijn. Door WebJail te baseren op Caja en Self-Protecting JavaScript kan dit nadeel hoogstwaarschijnlijk weg gewerkt worden.

Referenties

[1] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. WebJail : Least-privilege Integration of Third-party Components in Web Mashups Categories and Subject Descriptors. *Security*.

[2] Google. Google code. http://code.google.com/intl/nl/caja/docs/.

[3] Jonas Magazinius, Phu H Phung, and David Sands. Safe wrappers and sane policies for self protecting javascript. *The 15th Nordic Conf in Secure IT Systems*, 2010.

[4] Leo A Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 481–496, Washington, DC, USA, 2010. IEEE Computer Society.

[5] M.S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. 2008.

[6] P.H. Phung. *Lightweight Enforcement of Fine-Grained Security Policies for Untrusted Software*. PhD thesis, 2011.

[7] Phu H Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

Bibliografie

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

[2] Secure ecmascript. http://wiki.ecmascript.org/doku.php?id=ses:ses.

[3] Twitter. http://twitter.com.

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

[5] Adbrite exchange. http://www.adbrite.com/, 2011.

[6] Bidvertiser. http://www.bidvertiser.com/, 2012.

[7] S. V. Acker, P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail : Least-privilege Integration of Third-party Components in Web Mashups Categories and Subject Descriptors. *Security*.

[8] D. Crockford. The application/json media type for javascript notation (json). http://tools.ietf.org/html/rfc4627.

[9] X. Dong, M. Tran, Z. Liang, and X. Jiang. Adsentry: Comprehensive and flexible confinement of javascript-based advertisements.

[10] Google. Google code. http://code.google.com/intl/nl/caja/docs/.

[11] Google. Google maps. http://maps.google.com/.

[12] Google. *ig*oogle. http://www.google.be/ig.

[13] Google. Adsense crawler. http://support.google.com/adsense/bin/answer.py?hl=nl&answer=99376, 2012.

[14] Google. Google adsense. www.google.com/adsense, 2012.

[15] R. Hansen. Xss (cross site scripting) cheat sheet esp: for filter evasion. http://ha.ckers.org/xss.html, 2012.

[16] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society.

[17] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. *The 15th Nordic Conf in Secure IT Systems*, 2010.

[18] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 481–496, Washington, DC, USA, 2010. IEEE Computer Society.

[19] Microsorft. Microsoft adcenter. https://adcenter.microsoft.com/, 2012.

[20] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. 2008.

[21] M. D. Network. Proxy. https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Proxy.

[22] P. Phung. *Lightweight Enforcement of Fine-Grained Security Policies for Untrusted Software*. PhD thesis, 2011.

[23] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[24] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. Adjail: Practical enforcement of confidential and integrity policies on web advertisements.

Fiche masterproef

Studenten: Lynsey Hens
Brenda Lissens

Titel: Veilige integratie van JavaScript advertisements door middel van Secure ECMAScript

Engelse titel: Secure integration of JavaScript advertisements using Secure ECMAScript

UDC: 681.3

Korte inhoud:
Vele inkomsten in de online wereld worden gegenereerd door het opnemen van advertenties in websites. Deze advertenties worden met behulp van JavaScript geïntegreerd in de pagina van de publiceerder. Deze laatste heeft echter geen idee over wat deze scripts juist zullen uitvoeren binnen de context van zijn webpagina. Deze kunnen dus ook evengoed kwaadaardige intenties hebben waardoor bijvoorbeeld persoonlijke informatie dreigt vrijgegeven te worden. De bedoeling van deze thesis is om deze scripts op een gecontroleerde manier uit te voeren zodat de publiceerder geen veiligheidsrisicoÕs meer loopt. Er bestaan al heel wat technieken om dit doel te verwezenlijken, elk met hun eigen voor- en nadelen. In deze thesis hebben we geprobeerd om een bestaande oplossing, namelijk WebJail, te verbeteren. Bij WebJail moest de browser immers intern gewijzigd worden om te kunnen voldoen aan de vereiste veiligheid. Hierdoor was er gebruikersondersteuning nodig wat een enorm nadeel vormt. Met behulp van SecureECMAScript hebben we een prototype kunnen ontwikkelen waarbij browserwijzigingen niet langer nodig zijn. Tot slot vergelijken we het ontwikkelde prototype met het prototype gebaseerd op de bestaande techniek AdJail.

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

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

Assessoren: Prof. dr. ir. H. Blockeel
Dr. P. Philippaerts

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