SAPIENZA
UNIVERSITÀ DI ROMA

Firefox WebExtensions security infrastructure

Sapienza Università degli studi di Roma
Laurea triennale in Ingegneria Informatica

Francesco Pasquali
ID number 1933764

Advisor
Prof. Emilio Coppa

Academic Year 2020/2021

Thesis not yet defended

Firefox WebExtensions security infrastructure
Tesi triennale. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: pasquali.1933764@studenti.uniroma1.it

Contents

Chapter 1

Introduzione

Nota: *Perché i browser?*
I Web Browser sono programmi complessi volti a visualizzare contenuti di vario genere provenienti dalle fonti piu disparate, dai file locali alle risorse accessibili nell' internet, dalle immagini ai linguaggi di scripting. In un mondo sempre piu connesso, i browser sono diventati de facto applicazioni necessarie nella vita di tutti i giorni e utilizzate da una vastissima comunità di utenti; la loro diffusione unita alla capacità di eseguire codice javascript direttamente sulla macchina dell'utente rende i browser dei bersagli di interesse per il red-team, aprendo alla possibilità di compromettere un grande numero di dispositivi.

Nota: *Perché Firefox?*
Firefox è un Web Browser che in passato ha avuto un'ampia comunità di utenti, noto per le politiche a tutela privacy e la storica concorrenza con Internet Explorer.

Nota: *Perché le estensioni?*
Perché è codice scritto da terze parti a cui il browser da accesso ai contenuti web e ad API privilegiato normalmente non accessibili dal DOM, queste caratteristiche mi hanno indotto ad astrarre le estensioni come una sorta di ponte tra la pagina e il sistema, certo le estensioni non sono le uniche componenti del browser a svolgere questo ruolo (basti pensare all'interprete html), ma a differenza delle altre esse sono programmabili (e programmate) da sviluppatori esterni al progetto Firefox e potenzialmente ignari sulle misure di sicurezza nello sviluppo web in ambiente browser. Con questi presupposti ho ipotizzato uno scenario in cui l'attaccante abbia la possibilità di sfruttare una estensione vulnerabile ad attacchi di tipo html injection e sono andato a studiare se e in che modo il browser sia in grado di difendersi da tale scenario.

Qui forse dovresti citare le tecnologie di base dietro ad un browser: HTML, CSS, Javascript, etc.

direi una mezza frase su cosa e' Javascript

Direi non solo red team ma anche in generali attaccanti

E Chrome? Non siamo negli anni 2000!

Immagino che qui devi ancora espandere per dire che lo hai considerato... e che poi hai scelto di considerare le sue estensioni.

Mi raccomando cerca

Chapter 2

Concetti fondamentali

Il capitolo fa un escursus su definizioni ed elementi utili per la comprensione dei capitoli che seguiranno, nella Sezione 2.1 si introducono i tre linguaggi fondamentali del web, HTML5, CSS e Javascript al quale dedico la sezione Sezione 2.2. CSS verrà soltanto menzionato poiché non ha avuto rilievo nella mia ricerca. Seguono due sezioni di presentazione alle estensioni browser, Sezione 2.4 browser è un rapido riassunto sulla storia delle estensioni mentre Sezione 2.5 spiega la struttura di una WebExtension.

Aggiungi riferimenti alle nuove sezioni 2.3 e 2.6

e degli attacchi alle estensioni

2.1 I linguaggi fondamentali del World Wide Web

Un Web Browser è un'applicazione capace di richiedere e visualizzare risorse ottenute dalla rete all'interno di una finestra. Ad oggi un browser deve necessariamente essere in grado di interpretare questi tre tipi di risorsa per rendere il World Wide Web fruibile all'utente: documenti HTML5, fogli di stile CSS e script Javascript; tutti e tre sono living standards, cioè standard che modificano e aggiornano di anno in anno le proprie specifiche. HTML5 è un linguaggio di markup che descrive la struttura di una pagina web. Gli elementi della pagina sono rappresentati da tag HTML che talvolta modificano i metadati, la visualizzazione e il comportamento del documento; per esempio il contenuto del tag `<script>` può essere interpretato come codice Javascript ed eseguito al volo dal browser, ma lo stesso tag `<script>` può anche importare uno script da un URL o file sorgente. Il frammento di codice 2.1 è un esempio di utilizzo del linguaggio HTML5, in queste righe viene creato un documento che mostra a schermo la scritta "Archive:" ed include tre altre risorse: il tag `<link>` include e applica alla pagina il foglio di stile `page.css` mentre i due tag `<script>` caricano ed eseguono due script Javascript. In Firefox accanto ad HTML5 è presente un altro linguaggio di markup chiamato XUL usato per la UI del browser stesso.

due parole in piu su XUL

```
1  <!doctype html>
2  <html>
3      <head>
4          <meta charset="utf-8" />
5          <link rel="stylesheet" href="page.css" />
6          <script src="../shared/archive.js"></script>
7          <script defer src="sbMain.js"></script>
```

```
8                 </head>
9                 <body>
10                    <p>Archive:</p><br/>
11                    <section id="output"></section>
12                </body>
13            </html>
14
```

Listing 2.1. Un semplice documento html

2.2 Javascript

Javascript è il linguaggio principe della programmazione web lato frontend. Apparso per la prima volta nel 1995 in grembo a Netscape, la sua diffusione ha segnato l'inizio di un World Wide Web orientato alle applicazioni; ad oggi è usato in numerosi contesti diversi oltre alle pagine web, come server in Nodejs o reti neurali in Tensorflow, parti del browser stesso sono programmate con Javascript . È un linguaggio ad oggetti debolmente tipizzato che supporta la programmazione ad eventi, funzionale, imperativa e la programmazione ad oggetti basata su prototipi [2]. Rispetto ad altri linguaggi fortemente orientati alla programmazione oggettuale come Java, in Javascript gli oggetti si comportano come dizionari, il loro insieme di proprietà può venire espanso, ridotto e modificato a runtime. Tra le proprietà di un oggetto alcune sono rilevanti ai fini del linguaggio:

- `.valueOf()` Una funzione che ritorna il "valore" di un oggetto, viene invocata silenziosamente dall'interprete quando l'oggetto è sottoposto ad operazioni logico/matematiche;
- `.toString()` Una funzione che ritorna la rappresentazione dell'oggetto sottoforma di stringa.
- `.__proto__` È un oggetto che contiene i metodi e gli attributi ereditati, definisce un `.__proto__` a sua volta contenente metodi e attributi della super-super classe e coì via ricorsivamente. Quando si referenzia la proprietà di un oggetto dapprima la si cerca nell'oggetto stesso e se non viene trovata allora si esplorano ricorsivamente gli oggetti che formano la catena di `.__proto__` che termina con il `.__proto__` della classe base `Object`. È importante sottolineare che il sistema della catena di prototipi non è legato al tipo di oggetto che `.__proto__` referenzia, ma dalla sola presenza della chiave *__proto__* nell'oggetto esplorato e poiché si tratta di una proprietà come le altre è molto facile ridefinirlo.

prototype pollution?

- `.prototype` Usando la terminologia della programmazione ad oggetti, il `.prototype` di una classe contiene i metodi e gli attributi che verranno ereditati da ogni istanza della classe. In realtà in Javascript le classi non esistono ma al loro posto ci sono i costruttori, funzioni che creano, inizializzano e ritornano un oggetto che diremo "istanza" del costruttore. Il valore dell'attributo `.__proto__` dell'istanza, ovvero l'oggetto prototipale, è un riferimento al `.prototype` del costruttore.

- `.constructor()` È il riferimento alla funzione costruttrice dell'oggetto, questa proprietà si trova nel prototipo di un oggetto e viene solitamente usata per verificare l'appartenenza ad una "classe" di oggetti.

Il listato 2.2 è un esempio variegato dei concetti introdotti. Questo script dichiara un costruttore `Letter()` che assegna alle proprie istanze l'attributo `.number`, inoltre tutte le istanze di `Letter` ereditano il metodo `.getNumber()`, assegnato al prototipo del costruttore. Infine si assegnano alle variabili `a`,`b` e `z` tre diverse istanze di `Letter`. La figura 2.1 illustra gli oggetti discussi e i vari riferimenti ai prototipi, `Function.prototype` è il prototipo per tutti gli oggetti funzione mentre `Object.prototype` è il prototipo base di tutti gli oggetti.

```
1  function Letter(number) {
2      this.number = number;
3  }
4
5  Letter.prototype.getNumber = function() {
6      return this.number;
7  };
8
9  let a = new Letter(1);
10 let b = new Letter(2);
11 let z = new Letter(26);
12
```

Listing 2.2. Frammento di script, viene definito il costruttore `Letter` e lo si utilizza per creare tre oggetti

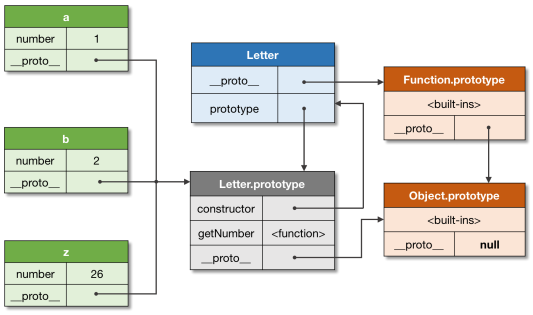

Figure 2.1

2.3 DOM

Il DOM è la rappresentazione di un documento HTML sottoforma di oggetti Javascript [1]. Attraverso l'interfaccia offerta dal DOM è possibile modificare programmaticamente gli elementi presenti nel documento; tra gli oggetti dichiarati nel DOM `document` rappresenta il documento vero e proprio mentre `window` rappresenta la finestra di visualizzaione che contiene il documento. `window` ricopre un ruolo molto importante nella programmazione web, esso è l'oggetto globale di tutti gli script eseguiti sulla pagina, ovvero un oggetto che ospita le variabili tra le sue proprietà, oltre agli oggetti built-in.

2.4 Breve storia delle estensioni

attacchi agli applet

Le estensioni browser non sono una novità. Già a partire dal 1995 Internet Explorer supportava lo sviluppo di plugin [3] affinché visualizzassero contenuti dinamici in un World Wide Web fatto di documenti statici, ma è a partire dal 1999 che il browser diventa programmabile quando Internet Explorer 4 iniziò a supportare la modifica dell'interfaccia utente. Il 2005 vede la nascita di Firefox e l'arrivo degli UserScript, piccoli script per la modifica delle pagine web installabili nel browser (precursori dei WebExtension Content Script approfonditi nelle prosime sezioni). Lo sviluppo di estensioni cross-browser fu per molto tempo una sfida, non c'era uno standard comune e ogni browser esponeva questa o quella funzionalità in piu rispetto agli altri; quando uscii, Google Chrome non era da meno, ma negli anni la sua ampia diffusione impose lo standard de-facto sulla scena; cercando di rimanere al passo con i tempi, Firefox si modificò per supportare le estensioni di Google Chrome introducendo il WebExtension framework nel 2017. Nonostante le differenze, ad oggi i browser stanno convergendo verso lo stesso formato di estensioni, non solo Firefox ma anche Opera e Safari si sono adattati allo standard e sviluppare addon cross-browser è diventato possibile.

2.5 Anatomia di una Web Extension

Una estensione è un insieme di script, fogli di stile e documenti html raccolti in una cartella o archivio compresso ottenibile dallo store del browser o dai file locali. L'installazione di una estensione può essere:

- **built-in**, pre-installata nel browser per migliorare la User Experience o integrare funzionalità utili all'applicazione, l'estensione WebCompat è un esempio; si tratta di una estensione built-in Firefox usata per introdurre fix di compatibilità dopo il rilascio di una nuova versione del browser.

- **temporanea**, rimane operativa fintanto che il browser è in esecuzione, ma dovrà essere reinstallata ad ogni avviamento dell'applicazione.

- **persistente**, persiste al riavvio dell'applicazione e viene riattivata ad ogni esecuzione del browser.

In ogni estensione che faccia uso del WebExtension framework deve essere presente il file `manifest.json` contenente un oggetto `JSON`. Le proprietà dell'oggetto `manifest.json` dichiarano i metadati dell'addon (come nome, versione e descrizione) e le risorse che verranno eventualmente utilizzate. Il `manifest.json` può contenere riferimenti a file di altro tipo che modificheranno il comportamento e l'aspetto dell'applicazione e si suddividono in:

- **background** Sono file che rispondo a eventi del browser, vengono chiamati di background perché sono eseguiti in modo silenzioso e operano con le componenti interne dell'applicazione. Questi script non possono accedere direttamente alle pagine web.

- **sidebar, popup e option page** Sono delle interfacce utente introdotte dall'estensione che vengono visualizzate sullo schermo in modo più o meno permanente. Le *sidebar* appaiono sul lato della finestra e rimangono visibili fino alla chiusura manuale da parte dell'utente, i *popup* sono piccole interfacce grafiche disegnate alla pressione di un bottone sulla toolbar o sulla address-bar. La *option-page* è mostrata quando l'utente accede alla pagina di modifica delle preferenze dell'estensione.
- **content script** Sono script usati per accedere e manipolare le finestre delle pagine web. Si potrebbe dire che sono la controparte UI dei *background* script ma a differenza di questi ultimi non possono interagire con le componenti interne del browser. I *content script* vengono eseguiti solo sulle pagine che rispettano i criteri di origine specificati nel `manifest.json` . A differenza delle pagine web i *content script* possono effettuare richieste cross-domain e usare un piccolo sottoinsieme del WebExtension API.
- **Web accessible resources** Sono risorse di qualsiasi tipo rese accessibili agli altri script dell'estensione.

La figura 2.2 illustra quali file possono essere referenziati per i vari elementi descritti sopra.

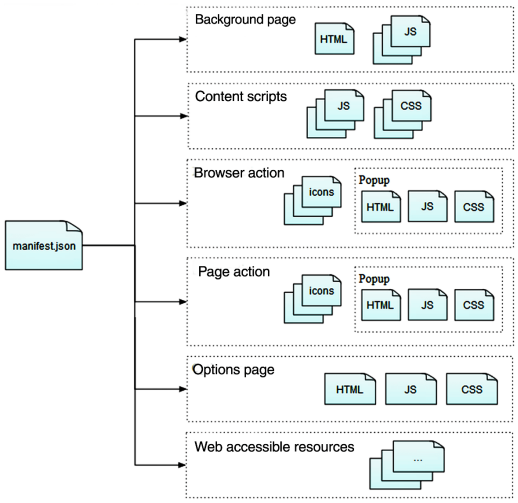

Figure 2.2. Tipi di file referrenziabili dal `manifest.json`

2.6 Il WebExtension API

Ogni script Javascript può fare uso di funzioni e oggetti messi a disposizione dall'ambiente in cui sta venendo eseguito, poiché ogni script ha diverse necessità, l'accesso a queste risorse può venire consentito o negato a seconda di quanto il sistema consideri rischioso esporre l'API ad un uso scorretto. Uno script di background deve necessariamente poter agire sugli eventi riguardanti il browser nel suo insieme, ma non si può dire lo stesso di uno script eseguito in una pagina web, interessato soltanto gli

eventi in atto sul suo documento; non a caso l'ambiente di esecuzione di background permette di modificare il comportamento del browser ma non permette di navigare verso altri URL e specularmente l'ambiente delle pagine web può richiedere un sottoinsieme di URI ma non ha alcun accesso agli API del browser.
Il WebExtension framework include nei suoi ambienti l'oggetto `browser` (e li suo alias `chrome`), esso raccoglie un insieme di api utili alle le estensioni e normalmente non accessibili dalle pagine web; quali siano questi API dipende dal tipo di script eseguito e dalla voce `"permissions"` del `manifest.json` . Tra gli api troviamo: `.runtime` [4] fornisce informazioni riguardo l'ambiente di esecuzione e offre funzioni di messaging,

continua

Chapter 3

Scenario e ambiente sperimentale

3.1 Threat model

- **Attaccante** Un membro del red-team interessato a compromettere il web browser (Firefox) della vittima come parte di una catena di attacco. Il suo obbiettivo è ottenere esecuzione di codice Javascript arbitrario che abbia accesso agli API interni di Firefox. Esso non ha altro modo di interagire con il sistema attaccato se non tramite una applicazione web di cui controlla quantomeno il codice dell'interfaccia utente; tale applicazione è visitabile attraverso un dominio pubblico ed espone endpoint http e/o https.
- **Vittima** Un utente privo di competenze in ambito cybersecurity. Esso è stato persuaso ad utilizzare la propria installazione di Firefox per visitare il dominio malevolo controllato dall'attaccante.

3.2 Specifiche applicazione e piattaforma

Per la mia ricerca ho svolto i test utilizzando l'ultima versione corrente di Mozilla Firefox v122.0 installata su MacOS Monterey v12.5 con processore Apple M1. Il codice sorgente studiato è stato scaricato dalla repository github ufficiale di mozilla, commit b59eed0; ogni menzione contenuta in questo articolo è relativa allo stato del codice risalente al suddetto commit. link: `https://github.com/JackieSpring/firefox/tree/b59eed054bcd27fbdf7e796ee5993dfb69d47f55`. L'applicazione è stata compilata seguendo le istruzioni riportate sul sito ufficiale di mozilla, link: `https://firefox-source-docs.mozilla.org/setup/macos_build.html`.

L'installazione di Firefox utilizzata dalla vittima è l'ultima versione corrente del browser (attualmente Firefox 122.0). La vittima ha installato una estensione, vulnerabile ad attacchi di code injection, che chiameremo "*vuln*".

3.3 L'estensione *"vuln"*

Il manifesto dell'estensione *"vuln"*dichiara tre componenti:

- Un **Content script** incaricato di ottenere coppie chiave-valore e inviarle in background. La coppia viene cercata all'interno della pagina e puo anche essere ricevuta attraverso un evento custom "trigger", in entrambi i casi la protezione Xray deve essere attenuata esponendo l'estensione.
- Un **Background script** memorizza la coppie ricevuta dal content script in un archivio salvato nel local storage del browser, implementato come array di oggetti.
- Una **Sidebar** che visualizza i dati delle coppie nella propria finestra inserendoli come testo html.

I primi due compongono la logica dell'estensione mentre la sidebar funge da front-end ma non è l'unica componente visualizzabile, insieme ad essa il manifest permette di esporre altre finestre UI che sono i popup e la pagina delle opzioni personalizzata; per restringere il campo di studio ho deciso di usare solo una delle tre finestre front-end, ho ritenuto la sidebar il giusto candidato per le proprietà del componente:

- Il compartimento Javascript in cui viene eseguito il codice della sidebar ha gli stessi permessi di accesso al WebExtensionAPI dello script di background soddisfacendo gli obbiettivi di bersaglio del threat model.
- Il compartimento della finestra viene creato all'apertura della sidebar e rimane attivo fino alla chiusura della stessa garantendomi controllo sulla durata del suo ciclo di vita; inoltre la sidebar rimane visibile accanto ad ogni tab aperta favorendone il debugging;

3.3.1 La vulnerabilità

La vulnerabilità risiede nella logica della sidebar, nella funzione `updateArchiveView`

```
1                keyField.innerHTML = data.key + ": ";
2                valField.innerHTML = data.value;
3
4                container.appendChild( keyField );
5                container.appendChild(valField);
6
```

Listing 3.1. Frammento di codice vulnerabile nella sidebar di *"vuln"*

quando gli elementi `keyField` e `valField` vengono aggiunti all'albero di nodi del DOM, il campo `.innerHTML` viene valutato come valido codice HTML5 e interpretato, se "key" o "value" dovessero contenere tag html essi verranno inseriti nel documento come elementi validi (l'unica eccezione è il tag `<script>`, questo caso verrà approfondito in seguito).

3.4 Workflow dell'estensione

Nota: *Questa sezione andrebbe spostata nel capitolo di spiegazione dell'attacco* Il content script ottiene dal WebContent un oggetto contenente le proprietà "key"

e "value", poiché l'oggetto proviene da un compartimento con privilegi minori il content script attenua la protezione sull'oggetto per poter leggere le proprietà non native, i dati vengono estratti, accorpati ad un messaggio, serializzati e trasmessi sul sistema di messaggi del runtime; durante questo passaggio vengono perse proprietà e valori non serializzabili, come le funzioni. Alla ricezione di un messaggio lo script in background deserializza l'archivio

Chapter 4

L'infrastruttura di sicurezza Firefox

Ogni sito internet moderno ha in qualche modo il bisogno di interagire con la rete, con uno storage locale, con il file system e con dispositivi audio e video, tutte risorse gestite dal sistema operativo, pertanto ogni sito deve poter interagire con il sistema operativo della macchina client ma dare questo livello di accesso a codice insicuro significa esporre il sistema ad alti rischi di sicurezza per queste ragioni il browser deve esporre API che rendano possibili le operazioni richieste dal sito senza però compromettere la macchina host.
Firefox non è direttamente responsabile della sicurezza del sistema, questo aspetto è invece gestito da rendering engine Gecko di cui Firefox è il front-end. Gecko applica quanto detto separando il codice eseguito in compartimenti logici, isolati in processi distinti, realizzando layer di separazione a livello applicativo e a livello fisico di sistema operativo.

4.1 Modello processi

Il codice che compone Firefox e Gecko non è eseguito sotto un unico processo, diversi servizi sono eseguiti in diversi processi per garantire solidità nel caso di fallimenti o compromissioni esterne; si dividono in tre categorie:

- **Parent Process** è il processo principe nonché padre di tutti gli altri, è incaricato di coordinare i processi figlio e gestire la comunicazione tra di essi. Visualizza pagine ad alti privilegi come `about:preferences` e `about:config`, pertanto ospita un ambiente di esecusione ristretto.

- **Helper Processes** sono processi che ospitano servizi, tra essi vi sono servizi di interazione con il file system, con la rete, con le immagini e altri ancora.

- **Content Processes** sono processi usati per renderizzare contenuto web, insieme al Parent sono gli unici a poter eseguire codice Javascript. Vengono suddivisi in "remote-type", proprietà che specificano i privilegi di accesso agli API.

Ci sono molti tipi di Content Process di cui due sono di particolare interesse per la mia ricerca:

- **WebExtensions Content Process** È utilizzato per caricare pagine in background e i subframe delle estensioni web; esiste una sola istanza di questo processo ed ha assegnato il remote-type "extension" che garantisce l'accesso al WebExtensionAPI e alla Shared Memory. Tutte le estensioni condvidono questo processo e sono visibili tra di loro.

- **Isolated Web Content Process** Sono usati per ospitare contenuti web attribuiti ad un sito, il codice web eseguito in questi processi è considerato insicuro e l'accesso diretto agli API di sistema non è permesso. Un nuovo web content viene allocato per ogni sito visitato su una browser tab, qualsiasi dominio visitato su una tab differente produce un nuovo processo Web Content isolato, invece subframe aperti sullo stesso dominio del superframe contenitore vengono eseguiti nello stesso processo del super-frame.

4.2 Sicurezza livello script

Il codice Javascript eseguito da Gecko non proviene solo da fonti terze come pagine web o estensioni, l'interfaccia grafica del browser (Firefox) e la logica sono controllati da moduli javascript ad alti privilegi di accesso pertanto gli script web non possono eseguire nello stesso ambiente del javascript di sistema. Il modello processi in se potrebbe sembrare una soluzione adeguata, ma se script web e di sistema dovessero eseguire nello stesso processo si creerebbe un conflitto, inoltre il browser deve poter accedere a oggetti del web content; la separazione in processi è troppo restrittiva per questi utilizzi; inoltre Javascript è un linguaggio a tipizzazione debole, funzionale e di cui le strutture di supporto alla programmazione ad oggetti sono modificabili, un esempio dei rischi introdotti da questa dinamicità sono gli attacchi di prototype pollution. Il modello di separazione dei processi non è sufficiente a gestire proprietà di linguaggio.

4.2.1 Security Policy

Una Security Policy è una definizione di che cose significhi "essere sicuro" per un sistema, nel caso di Gecko definisce il livello di accesso garantito verso un oggetto da parte di un altro oggetto in relazione a due rapporti: di Origine e di Privilegi. Gli oggetti dotati di stessa "origine" sono detti `"same-origin"` e hanno libero accesso alle proprietà, oggetti dotati di "origine" differente sono detti `"cross-origin"` e hanno accesso molto ristretto alle proprietà dell'altro.
Se l'oggetto acceduto si trova in uno scope di privilegio piu basso allora l'accedente avrà permessi di accesso libero ma potrà vedere solo un insieme ristretto di proprietà ma se invece l'oggetto acceduto dovesse trovarsi in uno scope con privilegi piu alti allora non otterà alcun privilegio di accesso. Script "privilegiati" possono clonare uno o piu oggetti in scope meno "privilegiati".

4.2.2 Same-Origin Policy

La Same-Origin Policy è un insieme di regole d'accesso a risorse situate su altre "origini". L' "origine" di una risorsa è definita come tripla di protocollo, dominio e porta, due risorse che condividono la stessa origine sono dette `same-origin`, altrimenti `cross-origin`. Le restrizioni imposte dalla Same-Origin Policy dipendono dal contesto d'uso:

- **Rete** Solitamente una risorsa di rete `cross-origin` ottiene accesso in scrittura ed embedding mentre la lettura viene proibita, cioé viene reso possibile inviare richieste cross-origin e incorporare risorse esterne ma non è possibile conoscere il contenuto della risposta. Le regole d'accesso di rete `cross-origin` possono essere modificate dalla risorsa acceduta tramite header http o tag html `meta`.
- **Storage** Gli spazi di archiviazione sono separati e indipendenti per ogni origine
- **Javascript API** Due sono gli oggetti visibili a `cross-origin`: `window` e `location`, di questi solo un sottoinsieme di proprietà è accessibile, tra queste sono notevoli `.postMessage` di `window` che consente di scambiare dati `cross-origin` tra gli script e `.href` di `location`, accessibile solo in scrittura, permette di redirigere la finestra.

Eccezioni

Non tutti i protocolli vengono trattati allo stesso modo dalla Same-Origin Policy, le risorse caricate da `about:blank` o `javascript:` sono considerate avere la stessa origine del documento che le contiene, mentre l'origine `file:///` è trattata come origine opaca cioè le risorse ottenute con questo protocollo non sono mai considerate same-origin, nemmeno se risiedenti nella stessa directory.

Iframe pitfall

Nota: *Rimuovere?* L'implementazione degli iframe risente di una piccola falla di referenza; quando viene inserito nel documento, `iframe` incorpora la pagina `about:blank` che viene sostituita non appena la risorsa è caricata, pertanto lo stesso iframe mostra agli API javascript oggetti con origine differente in due momenti diversi.

4.2.3 Compartimenti Javascript

I compartimenti sono aree di memoria indipendenti e sono alla base della sicurezza degli script in Gecko; ogni oggetto globale e gli oggetti associati alle sue proprietà condividono lo stesso compartimento. Gli oggetti memorizzati in un compartimento non sono direttamente accessibili da script appartenente ad un compartimento diverso, la condivisione di oggetti è ottenuta tramite oggetti wrapper memorizzati nel compartimento dello script che referenziano l'oggetto originale, il grado di accesso fornito dal wrapper verso l'oggetto rappresentato è determinato da Gecko secondo

la Security Policy. I criteri di origine sono valutati considerando come "origine" l'url dell'istanza di `window`, che è oggetto globale di ogni compartimento. I criteri di privilegio sono invece determinati secondo l'Entità di Sicurezza del compartimento.

- **Same-Origin** È il caso più comune, all'oggetto accedente viene concesso un wrapper trasparente che garantisce accesso completo all'oggetto richiesto come se fosse parte dello stesso compartimento.
- **Cross-Origin** Gecko assegna un wrapper cross-origin che limita l'accesso secondo la Same-Origin Policy.
- **Privilegio Maggiore** Se lo scope acceduto ha privilegi minori, allora si ottiene un wrapper Xray (di più su Xray Vision in seguito)
- **Privilegio Minore** Se lo scope acceduto ha privilegi maggiori, allora si ottiene un wrapper opaco che nega l'accesso all'oggetto.

4.2.4 Entità di Sicurezza

Un Entità di Sicurezza è qualunque entità che puo essere autenticata da un sistema, in Gecko esistono quattro entità sulle quali è definita una relazione di sicurezza. Per determinare il rapporto tra due entità si verifica se ciascuna sussume l'altra. Le quattro entità sono:

- **Entità di sistema** Supera ogni check di sicurezza, sussume sè stessa e tutte le altre entità. I compartimenti che eseguono codice di sistema sono istanze di questa entità.
- **Entità di Contenuto** È associata ai contenuti web ed è definita dall'origine del contenuto, sussume ogni altra entità con cui condivida l'origine.
- **Entità Espansa** È definita come una lista di "origini" su cui si ha accesso completo, essa sussume ogni entità che abbia inclusa la propria origine nella lista ma non è sussunta da nessuna di esse. Un esempio di impiego delle entità espanse sono i Content Script delle estensioni che possono accedere al contenuto di più pagine ma non viceversa. In generale l'entità espansa è utilizzata per garantire permessi cross-origin allo script senza però renderlo entità di sistema.
- **Entità Nulla** Fallisce quasi tutti i check e non sussume sè stesso, può essere acceduto solo da un'entità di sistema.

Le entità non modellizzano solo il livello di privilegio del compartimento ma anche l'origine el il test di relazione fornisce le informazioni sufficienti a computare un wrapper secondo la Security Policy, infatti se un compartimento sussume l'altro allora deve avere privilegi pari o maggiori, se entrambi si sussumano allora sono `same-origin`, se nessuno sussume allora si tratta di un accesso `cross-origin` se invece è uno solo dei due a sussumere allora vi è una differenza di privilegi. L'algoritmo di decisione impiegato da Gecko è rappresentato in questo grafico:

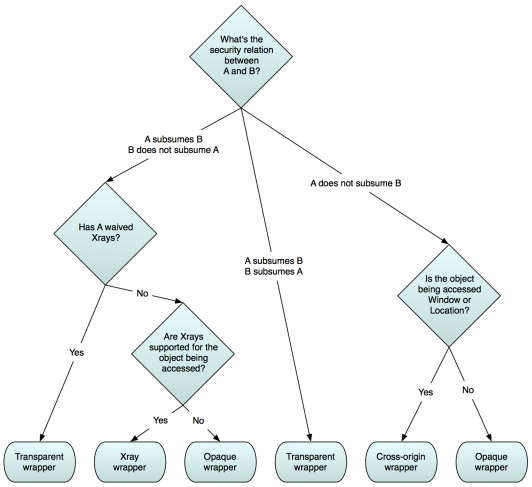

Figure 4.1. Computare un wrapper

4.2.5 Xray Vision

Javascript è un linguaggio molto malleabile il che lo rende imprevedibile in una contesto di sicurezza, oggetti provenienti da compartimenti insicuri potrebbero venire adeguatamente modificati per ingannare gli script privilegiati ad eseguire codice malevolo; per arginare il problema Gecko fa uso dei wrapper Xray, che consentono accesso completo alla forma "base" dell'oggetto ignorando le modifiche attuate dagli script, il modo in cui viene ottenuta dipende dall'oggetto acceduto.
Gli elementi del DOM sono gli oggetti piu comuni e hanno due implementazioni: una a livello Javascript che "vive" all'interno del proprio compartimento e memorizza lo stato corrente dell'oggetto, e una in codice nativo C++ che descrive la forma base dell'oggetto, quando il codice privilegiato deve accedere ad un elemento del DOM, gli viene restituito un wrapper Xray che mostra le proprietà della rappresentazione nativa. Alcuni oggetti esistono solo nel runtime Javascript, come gli `Array` o le `Promise`, allora si restringono le loro proprietà trattandoli come dizionari: metodi, getter e setter sono ignorati per impedire l'esecuzione di codice malevolo mentre il prototype di `Object` e `Array` viene rimpiazzato con il prototipo standard garantendo l'integrità.
Semmai il codice privilegiato avesse bisogno di conoscere lo stato corrente dell'oggetto la visione Xray può venire attenuata programmaticamente accedendo alla prorpietà `.wrappedJSObject`, ma a questo punto non si avrebbero piu garanzie di sicurezza su di esso, nè sui "figli".

Chapter 5

WebExtension nel dettaglio

Nota: *Approfondire il ruolo di* `WebExtensionPolicy` *nelle chiamate IPC*
Nota: *Approfondire il ruolo di* `ContentSecurityPolicy` *nelle chiamate agli API*
Quanto segue descrive la logica e le strutture che implementano il sistema delle WebExtensions a runtime, dal termine dell'installazione all'iniezione del codice nel processo. Sono operazioni svolte in due ambienti, nativo e javascript, ed interessano più componenti di Gecko, il mio studio si è concentrato sulla rappresentazione in ambiente javascript delle estensioni attive in un processo tralasciando i dettagli di archiviazione dei dati o di allocazione dei compartimenti. **Assert:** *Continua, spiega che JS è malleabile e che puoi fare prototypepollution e che il codice è troppo grande per C++*

5.1 Breve introduzione a XPCOM

XPCOM Framework è un ambiente di sviluppo multi piattaforma che astrae elementi del sistema operativo, come la gestione della memoria, passing di messaggi e memoria condivisa, e fornisce interfacce di comunicazione tra linguaggi programmazione; è il collante fondamentale tra i processi e le componenti di Gecko. Tra le feature di XPCOM il sottosistema XPConnect offre un linguaggio per definire interfacce di programmazione chiamato `idl` (Interface Definition Language) e un linguaggio per definire protocolli di comunicazione inter-processo ed RPC chiamato `ipdl` (Inte-process comunication Protocol Definition Language); il file di definizione, sia esso `idl` o `ipdl` , viene compilato in rappresentazioni equivalenti nei linguaggi di destinazione mentre la logica è implementata dal programmatore usando uno dei linguaggi specificati, al momento XPConnect supporta C++, Rust, Python, Javascript e pochi altri, tutti impiegati nello sviluppo di Gecko.
Voglio aprire una parentesi di approfondimento su `ipdl` per fare chiarezza sul concetto di *Attore*, menzionato nelle sezioni successive. A differenza delle interfacce un protocollo ha bisogno di due entità, dette Attori, che utilizzino il protocollo per un fine di comunicazione, `ipdl` si riferisce ad essi come Parent e Child. Il Parent viene collocato sul processo che rimarrà attivo più a lungo, idealmente il genitore, mentre il Child è l'entità corrispondente attiva su un altro processo, idelamente figlio del primo. Durante la compilazione, XPConnect genera sia le strutture di rappresentazione dei protocolli che le interfacce di programmazione per gli attori,

implementate allo stesso modo delle interfacce `idl` .

5.2 Loading

La vita di una estensione è legata a due classi: `AddonManager` ed `ExtensionProcessScript` , la prima è una classe singleton collocata nel Parent Process e responsabile della archiviazione degli Addon e delle estensioni (prima di supportare lo standard WebExtension, Firefox veniva espanso usando gli Addon, `AddonManager` è stata modificata successivamente per adattarsi al nuovo standard; quella degli Addon è una storia affascinante che si intreccia con l'evoluzione del sistema di sicurezza del browser). Quando una estensione viene installata è `AddonManager` a prendersi carico del collezionamento dei nuovi script nel database e di avviarli per la prima volta mentre il browser è in esecuzione; ma se `AddonManager` è situata nella memoria del Parent Process come può iniettare uno script su altri processi isolati? `ExtensionProcessScript` è un'altra classe singleton attiva su ogni Content Process, è incaricata di ottenere i dati e il codice delle estensioni e farne il setup preparandoli per il successivo utilizzo; inoltre è indirettamente responsabile dell'avvio all'interno del processo. Per rappresentare l'estensione `ExtensionProcessScript` istanzia un oggetto `WebExtensionPolicy` che ha la duplice funzione di astrarre l'estensione stessa e le informazioni di sicurezza associate tra cui l'entità di sicurezza, la Content Security Policy, l'Origin e altre proprietà interne; tutti gli oggetti menzionati sono descritti da interfacce `idl` ed implementati in codice nativo, eccetto `mozIExtensionProcessScript.` scritto direttamente in Javascript.

5.3

Il codice del framework può essere trovato nella cartella `toolkit/components/extensions/`, è composto da molti file che suddividerò in tre categorie: core, integrativi ed implementativi. I file di core compongono

Chapter 6

Conclusioni

Bibliography

[1] *Introduction to the DOM*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

[2] *JavaScript*. URL: https://en.wikipedia.org/wiki/JavaScript#Creation_at_Netscape.

[3] Todd Schiller. *A Brief History of Browser Extensibility*. URL: https://medium.com/brick-by-brick/a-brief-history-of-browser-extensibility-bcfeb4181c9a.

[4] *WebExtension API runtime*. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/runtime.