Universitá
Ca' Foscari
Venezia

Corso di Laurea in Informatica

Ordinamento ex D.M. 270/2004

Tesi di Laurea

Web APIs Security

Relatore
Dott. Stefano Calzavara

Correlatore
Dott. Marco Squarcina

Laureando
Daniele Rigon
Matricola 857319

Anno Accademico
2018/2019

Alla mia famiglia.

ABSTRACT

Nel corso degli ultimi decenni la tecnologia è diventata sempre più presente nella quotidinità di ogni individuo, che riguardi lavoro, svago o vita quotidiana, e in particolar modo Internet. Come ogni cosa però presenta aspetti positivi e negativi e, essendo il World Wide Web un modo vastissimo, sono numerosi i rischi al suo interno. In questa tesi ho preso in esame uno degli infiniti argomenti che interessano il web, ovvero le Web APIs, uno strumento molto usato e che probabilmente alcuni di noi ne usufruiscono senza saperlo, spiegando e mostrando loro problemi legati alla loro sicurezza. La domanda di questa ricerca quindi è: *"Come faccio ad individuare i rischi di queste APIs e come posso affrontarli?"* Per rispondere a questa domanda ho studiato un piccolo set di APIs, mostrando per ognuna come possono essere usate secondo tre differenti threat model:

- un utente che usa le API per migliorare l'esperienza utente;
- un utente che usa tali API ma non sa che ci possono essere conseguenze negative (ad esempio viene inserito uno script malevolo nella pagina);
- un attaccante che usa le API in modo malevolo;

L'obiettivo di questo lavoro è stato quello di mostrare le vulnerabilità di ogni API presa in esame con la speranza di poter aiutare a migliorare i rischi legati alla sicurezza.

INDICE

CAPITOLO 1

INTRODUZIONE

Alla base di questo studio vi è l'analisi della sicurezza di un piccolo set di APIs [1], in particolare si pone l'attenzione sulle loro vulnerabilità spiegandole nella speranza che vengano risolte.
Le motivazioni che mi hanno spinto ad approfondire questo tema sono legate al fatto che ogni giorno mi vengono poste domande riguardo consigli, delucidazioni o dubbi riguardo a questo argomento, il più delle volte da persone che non hanno molto dimestichetta coi sistemi informatici.
L'obiettivo di questa tesi è fornire un'analisi di sicurezza accurata e cercare di fornire, per quanto basilari, delle soluzioni per prevenire o rimediare a vulnerabilità informatiche, con l'augurio che esse vengano risolte al più presto.
È stata condotta una ricerca individuale, supervisionata e corretta dal mio relatore, riguardo i problemi di sicurezza di ognuna delle tre APIs studiate.
La tesi è articolata in tre capitoli: nel primo viene spiegata la *Geolocation API* [1] la quale, utilizzata nei siti web o nelle web app, serve a tracciare la posizione dell'utente, con possibili rischi di problemi di privacy poichè un Web attacker potrebbe tracciare la posizione dell'utente e rubarla; nel secondo mi sono occupato della *PaymentRequest API* [2] la quale salva le informazioni dell'utente, sia personali che di pagamento, cosi da poterle riutilizzare automaticamente senza doverle reinserirle ogni qualvolta se ne ha bisogno, mostrando però che queste possono anche essere intercettate e rubate da un Web attacker; nella terza ed ultima parte, che è anche la più complessa, l'attenzione verte sui *Service Worker* i quali vengono usati principalmente per il caching in modo da migliorare l'esperienza utente, mostrando però anche come questi possano essere utilizzati in modo malevolo per rubare informazioni dell'utente intercettando le richieste di rete.
In ognuna di queste sezioni viene fatta una panoramica dell'API considerata, si entra nel dettaglio a spiegare le specifiche e i metodi principali e in ultima analisi vengono analizzati i problemi relativi alla sicurezza.
Per ognuna delle API studiate vi sono inoltre tre differenti threat model:

[1] Interfacce che gli sviluppatori e i programmatori possono utilizzare per espandere le funzionalità di programmi o applicazioni, ad esempio facendo interagire due programmi e/o piattaforme che altrimenti sarebbero state incompatibili tra loro (Google Maps mette a disposizione degli sviluppatori le proprie API da utilizzare nei loro programmi e/o piattaforme; Facebook dà la possibilità agli sviluppatori, tramite l'uso delle proprie API, di usare alcune delle proprie funzionalità per realizzare applicazioni da usare all'interno del social network, facilitando molto la programmazione.

- un utente che usa le API per migliorare l'esperienza utente;
- un utente che usa tali API ma non sa che ci possono essere conseguenze negative (ad esempio viene inserito uno script malevolo nella pagina);
- un attaccante che usa le API in modo malevolo;

Grazie a questo lavoro è stato possibile analizzare e scoprire i problemi legati a questi API e ad ottenere dei risultati che saranno poi esposti in modo dettagliato al termine di ogni capitolo e nelle conslusioni finali di questa tesi.
Tutto il codice presente nella tesi è reperibile al seguente repository GitHub [2].

[2] `https://github.com/danielerigon4/Tesi-Web-Api`

CAPITOLO 2

GEOLOCATION

2.1 Overview

La Geolocation API viene utilizzata per ottenere la posizione geografica di un utente. Poiché questo può compromettere la privacy la posizione non è disponibile a meno che l'utente non la approvi: su un dispositivo mobile avremo un set di coordinate provenienti dal sensore GPS mentre su un portatile potremo usare il posizionamento legato all'ip della connessione internet.

2.2 Specifiche

2.2.1 Oggetto della geolocalizzazione

Nella Geolocation API la posizione viene resa nota utilizzando la proprietà *navigator.geolocation* [3] presente nell'oggetto *Navigator*, un'interfaccia che rappresenta lo stato dell'utente. Se l'oggetto esiste, il servizio di geolocalizzazione è disponibile. Per testare l'esistenza di tale oggetto, così da avere disponibili i metodi di geolocalizzazione, bisogna verificare se l'oggetto esiste, nel modo:

```
1 if ("geolocation" in navigator) {
2   /* la geolocalizzazione e disponibile */
3 }
4 else {
5   /* la geolocalizzazione non e disponibile */
6 }
```

2.2.2 Metodi

Ci sono solamente tre metodi a disposizione: *getCurrentPosition*, *watchPosition* e *clearWatch*. I primi due sono utili a ottenere la posizione corrente mentre il terzo serve ad annullare la ricerca della posizione corrente. La differenza tra i primi due va ricercata nella loro periodicità, mentre il primo metodo fornisce il dato una sola volta, il secondo si attiva automaticamente ogni qualvolta la posizione cambi, o ogni tot intervallo di tempo.
La sintassi per invocare questi metodi è la seguente:

```
1 navigator.geolocation.getCurrentPosition(success, error, options);
2 navigator.geolocation.watchPosition(success, error, options);
```

dove è obbligatoria solamente la callback in caso di successo *success*, mentre gli altri due parametri sono opzionali.
La *successCallback* viene chiamata quando vi è un'operazione di successo alla chiamata di una nuova posizione. La *errorCallback* viene chiamata quando vi sono uno o più errori, e vengono gestiti all'interno di questo metodo. *Options* definisce i parametri in base ai quali vengono poi chiamate le callback di successo o fallimento. Un esempio concreto, preso da W3C [4], è questo:

```
1  /* Si richiede una posizione accettando solo posizioni non superiori a
       10 minuti che sono memorizzate nella cache.
2  Se non c'è un nuovo oggetto position nella cache verra invocata la
       callback d'errore */
3
4  navigator.geolocation.getCurrentPosition (successCallback, errorCallback
       , { maximumAge: 600000 , timeout: 0 });
5
6  function  successCallback (position){
7     /* Avendo specificato in "maximumAge" il range di 10 minuti la
       posizione è garantita al massimo per 10 minuti.
8     Usando un timeout di 0ms, se non c'è una posizione disponibile nella
       cache viene invocata la callback di errore
9     e non verra avviato nessun processo di acquisizione di una nuova
       posizione */
10 }
11
12 function  errorCallback (error){
13    switch (error.code) {
14      case error.TIMEOUT:
15        /* chiamata a funzione quando non esiste una posizione
       appropriata nella cache */
16        doFallback();
17        /* prende un nuovo oggetto position */
18        navigator.geolocation.getCurrentPosition (successCallback,
       errorCallback);
19        break;
20     case ... /* vengono trattati tutti gli altri casi di errore */
21   };
22 }
23
24 function  doFallback () {
25    /* Se nessuna posizione è disponibile nella cache allora sara
       richiamata una Fallback in una posizione predefinita */
26 }
```

GetCurrentPosition
Il metodo *GetCuttentPosition* viene chiamato per ottenere la posizione corrente dell'utente, fornendo il dato ogni volta che viene chiamata.

WatchPosition
Il metodo *WatchPosition* restituisce la posizione dell'utente ad ogni intervallo di tempo specificato.

ClearWatch
Il metodo *ClearWatch* viene chiamato per annullare il monitoraggio della posizione dell'utente.

Un esempio [5] dell'uso di questi metodi è il seguente:

```
1 function success(position) {
2   document.getElementById('latitude').innerHTML = position.coords.
     latitude;
3   document.getElementById('longitude').innerHTML = position.coords.
     longitude;
4   document.getElementById('position-accuracy').innerHTML = position.
     coords.accuracy;
5 }
```

Che produrrà la seguente pagina:

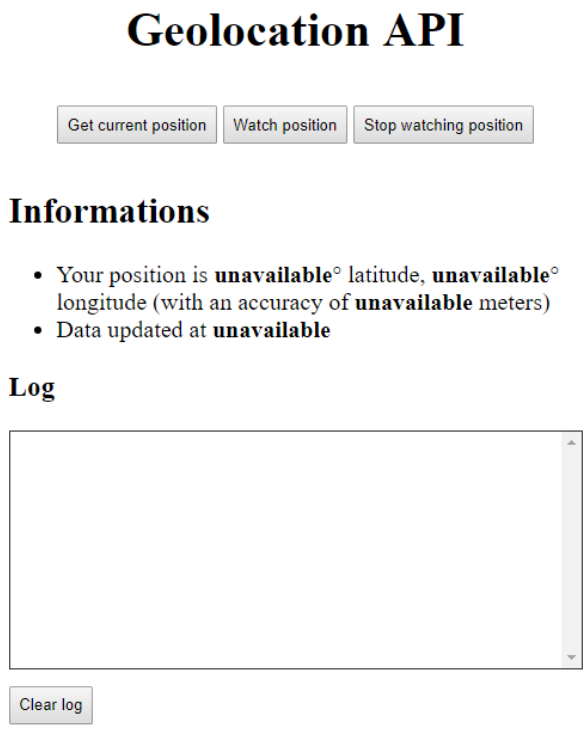

Figura 2.1: Pagina info Geolocation

Quando saranno chiamati i metodi *getCurrentposition* o *WatchPosition* verrà chiesto all'utente il permesso per usare la posizione.

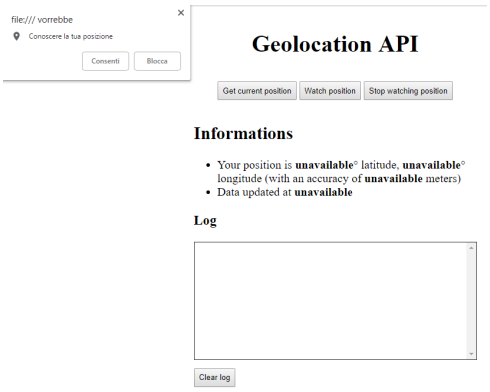

Figura 2.2: Richiesta permesso

Se l'utente rifiuta la posizione non viene calcolata e viene mostrato un messaggio di errore in console.

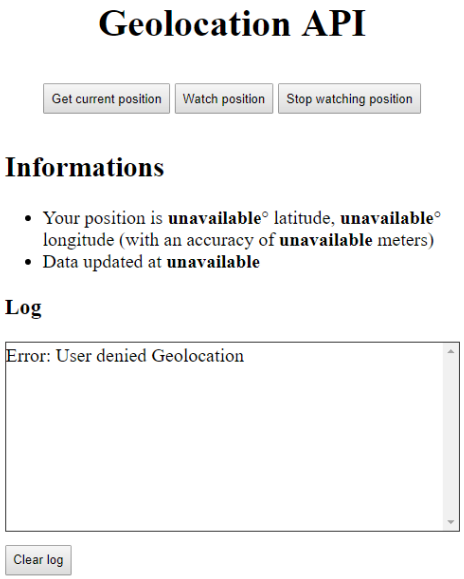

Figura 2.3: Rifiuto autorizzazione

Se l'utente acconsente all'utilizzo della posizione verrà mostrato un messaggio di conferma in console e saranno mostrate a video tutte le informazioni disponibili dell'utente.

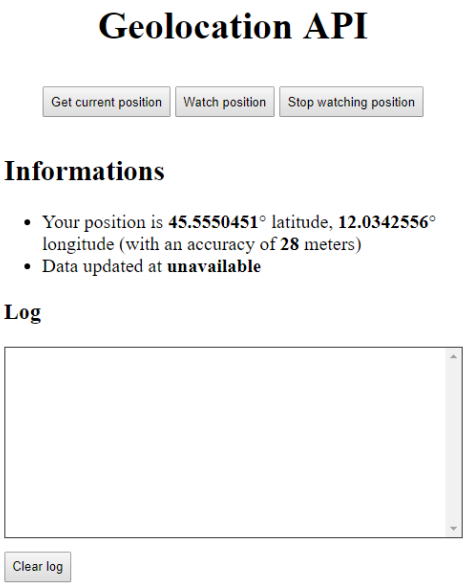

Figura 2.4: Accetto autorizzazione

2.3 Rischi e sicurezza

Uno dei principali problemi della Geolocation API [6] [7] è rappresentato dal fatto che un Web Attacker può rubare la posizione ad un utente che accetta di condividerla all'interno del sito attaccante ignaro del fatto che non sia sicuro, oppure attraverso un cross-site scripting (XSS) [1] all'interno di un sito considerato fidato. Gli oggetti per tracciare le coordinate (latitudine e longitudine) risiedono all'interno del DOM il quale è accessibile con JavaScript e attraverso il quale si possono prendere le informazioni sulla posizione dell'utente. Nella maggior parte dei casi succede che la posizione viene richiesta e condivisa in quanto viene chiesta in siti web affidabili. Un problema importante è che se l'utente non disabilita il tracciamento della posizione il browser continuerà a esporla all'attaccante.

[1] Tipo di vulnerabilità di sicurezza presente nelle applicazioni web. Questo attacco consente agli aggressori di inserire script sul lato client in pagine Web visualizzate da altri utenti, e questa vulnerabilità può essere usata dagli aggressori per bypassare i controlli di accesso, come ad esempio la same origin policy.

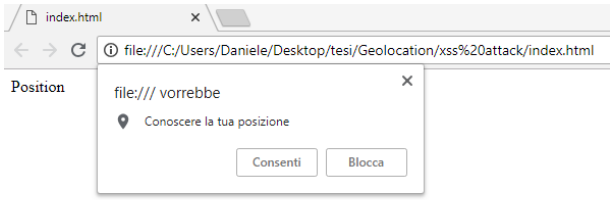

Figura 2.5: Richiesta posizione

Supponiamo che un utente malintenzionato abbia avuto modo di inserire una vulnerabilità XSS in un sito Web; tutto ciò che dovra fare è fare in modo che la vittima esegua il seguente codice JavaScript per rubare la posizione.

```
1  function showPosition(position){
2      var latitude = position.coords.latitude;
3      var longitude = position.coords.longitude;
4      var pos= "Latitude: " + latitude + "<br>Longitude: " + longitude;
5      document.getElementById("mydiv").innerHTML = pos;
6  }
7  function getLocation(){
8      navigator.geolocation.getCurrentPosition(showPosition);
9  }
10 getLocation();
```

Il codice utilizza le proprietà del DOM *cords.latitude* e *cords.langitude* per determinare rispettivamente la latitudine e longitudine, memorizzandole in una variabile, inviandole poi all'attaccante.

2.4 Compatibilità web

2.4.1 Desktop e mobile

Basic support	5	12	3.5	9	16	5	Yes	Yes	12	4	15	Yes	Yes
Secure context required	50	?	55	No	37	Yes	51 *	50	?	55	37	Yes	?
`clearWatch`	5	Yes	3.5	9	16	Yes	Yes	Yes	Yes	4	15	Yes	Yes
`getCurrentPosition`	5	Yes	3.5	9	16	Yes	Yes	Yes	Yes	4	15	Yes	Yes
`watchPosition`	5	Yes	3.5	9	16	Yes	Yes	?	Yes	4	15	Yes	Yes

Full support — No support
Compatibility unknown — See implementation notes.

Figura 2.6: Compatibilità Desktop e mobile

2.5 Conclusioni

Abbiamo analizzato le specifiche e i problemi di privacy di una delle API più semplici del Web che, come ogni tecnologia, può essere un'arma a doppio taglio. Tutti i threat model citati all'inizio della tesi possono essere identificati nel paragrafo. Come primo scenario consideriamo l'utente che usa l'API per migliorare l'esperienza utente: se guardiamo al commercio nel web, che negli ultimi anni ha preso sempre più piede, grazie alla geolocalizzazione gli utenti potranno avere sempre pubblicità mirata e personalizzata nei contenuti, della quale ne approfittano professionisti del marketing, rivenditori, enti governativi, forze dell'ordine, ecc; oppure possiamo considerare il fatto che grazie alla geolocalizzazione si possa ritrovare il proprio cellulare smarrito o rubato. Per passare al secondo threat model, ovvero un utente usa l'API ma non sa che ci possono essere conseguenze negative, possiamo prendere a titolo d'esempio un'azienda che, adottando tale tecnologia, potrebbe aumentare il rischio della privacy personale e dei dipendenti nel caso alcuni dati vengano rubati. Considerando l'ultimo scenario, ovvero un attaccante che usa le API intenzionalmente in modo malevolo. Questa persona potrebbe avere le informazioni di un tag GPS e di geolocalizzazione, in combinazione con altre informazioni personali, e usarle per identificare il luogo presente o futuro di un individuo, facilitando la possibilità di causare danni a persone e/o proprietà (furto, stalking, rapimento, violenza domestica, furto d'identità, ecc). Ovviamente il crimine fisico è meno prevalente del crimine informatico attraverso il quale, tramite i dati di geolocalizzazione IP, si può identificare la posizione fisica del computer dell'utente dal quale si possono rubare informazioni personali (indirizzo di casa, lavoro, scuola, itinerari giornalieri, carte di credito, ecc). Chiaramente, come detto precedentemente, il fattore di rischio si estende non solo all'individuo ma anche a imprese, dipendenti e via dicendo. Inoltre, quando qualcuno utilizza un'applicazione, potrebbero esserci più servizi che controllano i dati: il fornitore di tali servizi, i punti di accesso, gli sviluppatori, ecc. e molte volte gli utenti non riescono a capire l'origine della raccolta di tali dati. Questo fattore, assieme al modo in cui vengono utilizzati, con chi sono condivisi, se saranno traferiti altrove, per quanto saranno conservati e da chi, sollevano preoccupazioni agli utenti. Chiaramente esistono diversi modi per mitigare questi rischi, ad esempio attraverso soluzioni di salvaguardia tecnologica e/o attraverso la consapevolezza dell'utente di tali rischi.
Col passare del tempo ci saranno sempre nuove tecnologie e nuovi servizi, argomenti e temi ricorrenti che la società continuerà a considerare e discutere in positivo o negativo. Si potrebbe essere contenti di avere la vita semplificata per alcune tecnologie offerteci, come in questo caso per quanto riguarda la geolocalizzazione, ma al contempo si potrebbe essere preoccupati per la propria privacy: chi ha i miei dati? Come vengono usati? Sono protetti?

CAPITOLO 3

PAYMENTREQUEST API

3.1 Overview

La PaymentRequest API [8] nasce con l'intento di creare esperienze di pagamento semplificate, in quanto ogni sito web ha il proprio sistema di pagamento e molti siti richiedono la ridigitazione manuale delle stesse informazioni più volte, le quali possono essere invece memorizzate e riutilizzate dall'API per completare più rapidamente le transazioni online.

3.1.1 Vantaggi [9]

- **Esperienza di acquisto rapida:** gli utenti immettono i propri dati una sola volta nel browser, e dopo averli inseriti non è più necessario reinserirli su siti diversi;
- **Esperienza coerente su ogni sito che supporta l'API:** poiché la pagina di pagamento è controllata dal browser si può personalizzare l'esperienza utente, ad esempio includendo la localizzazione per impostare automaticamente la lingua preferita dell'utente o altre features;
- **Gestione delle informazioni:** gli utenti possono gestire le loro carte di credito e gli indirizzi di spedizione direttamente nel browser. Un browser può anche sincronizzare queste informazioni tra dispositivi, rendendo più semplice per gli utenti passare dal desktop al cellulare e viceversa quando si acquistano oggetti;
- **Gestione coerente degli errori:** il browser può controllare la validità dei numeri delle carte e può comunicare all'utente se una carta è scaduta o sta per scadere, può suggerire automaticamente quale carta utilizzare in base ai modelli di utilizzo passati o alle restrizioni del commerciante, o consentire all'utente di dire quale sia la carta predefinita/preferita;
- **Esperienza utente migliorata:** coerenza tra i siti Web, tra browser e sistemi operativi e nuove funzionalità del browser per semplificare il checkout, ecc;
- **Miglioramento della sicurezza:** la PaymentRequest API ha il potenziale per ridurre le opportunità di frode e può facilitare l'adozione di metodi di pagamento più sicuri. Purtroppo ci sono dei problemi di sicurezza analizzati al paragrafo 3.4;
- **Responsabilità inferiore:** in passato, per creare un'esperienza utente semplificata, i commercianti dovevano memorizzare le credenziali di pagamento degli utenti.

Questo non è più necessario, il che può aiutare a ridurre la responsabilità del commerciante nei confronti del cliente.

3.1.2 Come funziona

La PaymentRequest API consente a un utente di completare una transazione più facilmente riutilizzando le informazioni memorizzate nel browser o in app di pagamento di terze parti. Quando l'utente preme un pulsante in una pagina di checkout collegata all'API il commerciante utilizza l'API per richiedere il pagamento. Il commerciante fornisce informazioni su prezzo, valuta e un elenco di metodi di pagamento accettati, e può inoltre richiedere al browser di creare un'interfaccia utente semplificata per raccogliere l'indirizzo di spedizione, le informazioni di contatto e altri elementi all'utente. Il browser determina quali metodi di pagamento sono supportati dal commerciante tra le varie "app di pagamento" mostrandole all'utente. L'utente seleziona un'app di pagamento con la quale pagare, la quale può comportare ulteriori interazioni con l'utente (ad esempio per l'autenticazione). Al completamento l'app di pagamento restituisce i dati tramite l'API al commerciante.

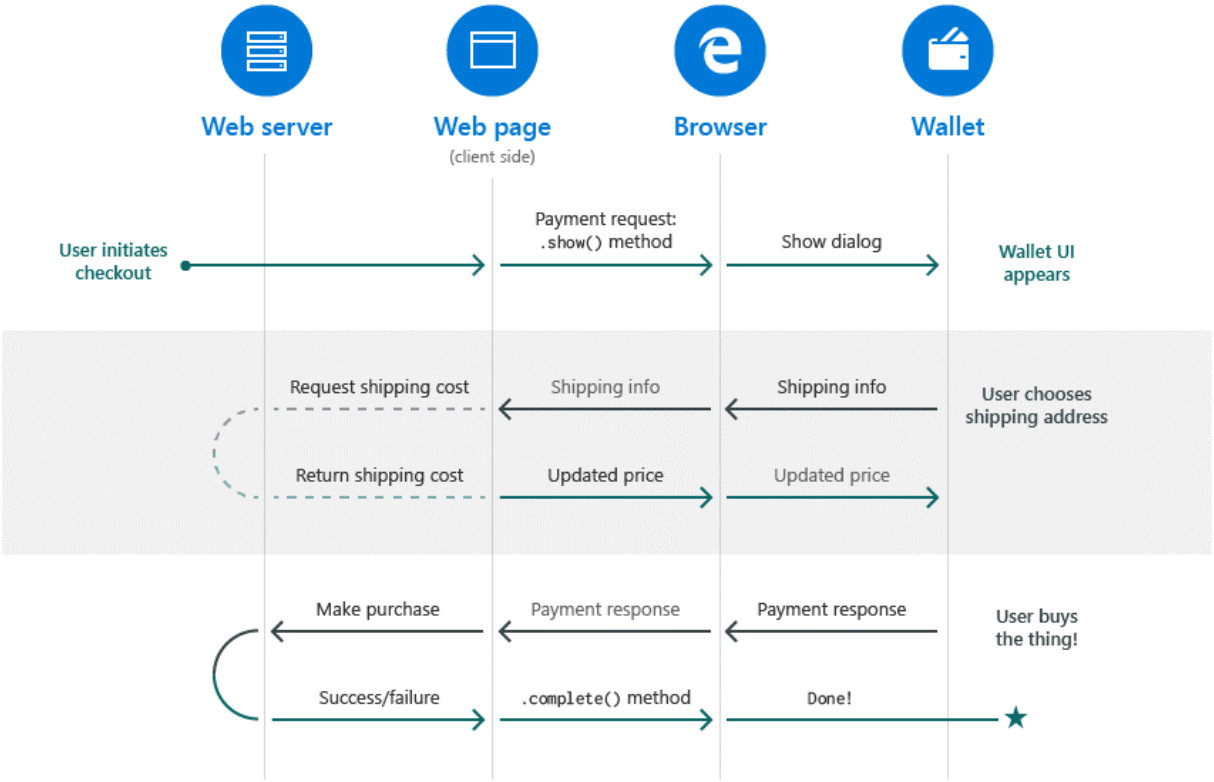

Figura 3.1: Schema Payment Request API [10]

3.1.3 Uso API

Ruolo dell'utente
Gli utenti beneficiano del riutilizzo delle informazioni inserite nel browser o nelle app di pagamento. Quindi, quando si visita un sito Web che sfrutta la PaymentRequest API gli utenti avranno l'opportunità di sfruttare il riutilizzo semplificato delle informazioni archiviate.
Ruolo del commerciante
L'API influisce sul front end (l'interfaccia dell'esperienza utente) e non sul back-end, pertanto il commerciante non dovrebbe dover apportare modifiche all'elaborazione back-end dei vari metodi di pagamento; questo sarà compito del fornitore della pagina di pagamento il quale sostituirà i moduli Web con le chiamate alla PaymentRequest API.
Ruolo del browser
Il browser svolge diversi ruoli:

- Calcola l'intersezione dei metodi di pagamento accettati dal commerciante e registrati dall'utente;
- Visualizza l'interfaccia utente che consente all'utente di inserire le proprie informazioni;
- Funge da canale per i dati da e verso il commerciante e da e verso l'utente.

App di pagamento
La PaymentRequest API determina se un'app di pagamento "corrisponde" a una determinata transazione definendo un algoritmo che considera i metodi di pagamento accettati dal commerciante, dichiarati attraverso un elenco di identificativi del metodo di pagamento, passati attraverso l'API. Al fine di proteggere la privacy degli utenti i commercianti hanno accesso a informazioni molto limitate dell'utente.

Vediamo in che modo la PaymentRequest API influisce sul flusso dei metodi di pagamento che già supporta. Il normale flusso per gli utenti di solito implica qualcosa del genere:

- Scansione di un elenco di metodi di pagamento accettati;
- Scelta di un metodo;
- Per i metodi di pagamento che prevedono il lancio di un'app o la visita a un sito Web si invia l'utente a quell'app o sito;
- Pagamento completato.

La PaymentRequest API consente un flusso migliorato:

- L'utente preme un pulsante di acquisto singolo;
- Il browser visualizza le app di pagamento dell'utente che possono essere utilizzate per la transazione, ed è probabile che i browser supportino le preferenze dell'utente in modo che un'app di pagamento venga avviata automaticamente su un determinato sito Web, semplificando il checkout.
- Per i metodi di pagamento che prevedono il lancio di un'app o la visita a un sito Web, inviare l'utente a quell'app o sito;
- Pagamento completato;

Differenze tra metodo di pagamento e app di pagamento
Un'app di pagamento è il software che l'utente utilizza per pagare, la quale può supportare uno o più metodi di pagamento e può essere implementata utilizzando diverse tecnologie. I browser possono anche fungere da app di pagamento, memorizzando le credenziali dell'utente. In generale più app di pagamento possono implementare lo stesso metodo di pagamento. Vi sono casi importanti in cui è disponibile una sola app di pagamento autorizzata a supportare un metodo di pagamento, mentre ci sono casi in cui più app di pagamento possono servire diversi metodi di pagamento. In questo caso non è il commerciante che deve preoccuparsi dell'integrazione software, ma deve solamente richiedere le informazioni attraverso la PaymentRequest API.

3.2 Specifiche

3.2.1 Metodi [2]

Per utilizzare l'API lo sviluppatore deve fornire e tenere traccia di una serie di informazioni chiave, le quali vengono passate al costruttore PaymentRequest come argomenti e successivamente utilizzate per aggiornare la richiesta di pagamento visualizzata all'utente. Queste informazioni sono:

- **PaymentMethodData:** rappresenta i metodi di pagamento che il sito supporta;

- **PaymentDetails:** rappresenta i dettagli della transazione. Ciò include il costo totale e facoltativamente un elenco di beni o servizi acquistati, beni materiali, opzioni di spedizione o "modificatori" su come vengono effettuati i pagamenti: ad esempio "se paghi con una carta di credito di tipo X incorre in una tassa di elaborazione di tot";

- **PaymentOptions:** il PaymentOptions viene passato al costruttore PaymentRequest e fornisce informazioni sulla consegna del prodotto: ad esempio per i beni fisici il commerciante avrà bisogno di un indirizzo fisico dove spedire, mentre per i beni digitali e sufficeinte un'e-mail. Una volta che il PaymentRequest è stato costruito viene presentato all'utente finale tramite il metodo show(), il quale ritorna una promise che, una volta che l'utente conferma la richiesta di pagamento, si traduce in una PaymentResponse;

- **PaymentRequest:** la PaymentRequest serve a effettuare una richiesta di pagamento, in genere associata all'avvio di un processo di pagamento da parte dell'utente. La PaymentRequest consente agli sviluppatori di scambiare informazioni con l'user agent mentre l'utente sta fornendo dati in input. Poiché la visualizzazione simultanea di più interfacce PaymentRequest potrebbe confondere l'utente, questa specifica limita lo user agent a visualizzarne uno alla volta tramite il metodo show();

- **PaymentDetailsInit:** Il PaymentDetailsInit viene utilizzato nella costruzione della richiesta di pagamento;

- **PaymentResponse:** un PaymentResponse viene restituito quando un utente ha selezionato un metodo di pagamento e approvato una richiesta di pagamento.

PaymentMethodData

PaymentMethodData [11] contiene gli identificativi dei metodi di pagamento accettati dal sito Web e qualsiasi dato specifico del metodo di pagamento associato.

```
const methodData = [
    {
        supportedMethods: "basic-card",
        data: {
            supportedNetworks: ["visa", "mastercard"],
            supportedTypes: ["debit", "credit"],
        },
    },
    {
        supportedMethods: "https://example.com/bobpay",
        data: {
            merchantIdentifier: "XXXX",
            bobPaySpecificField: true,
        },
    },
];
```

PaymentDetails

I *details* [12] contengono informazioni sulla transazione che l'utente è invitato a completare.

```
const details = {
    id: "super-store-order-123-12312",
    displayItems: [
        {
            label: "Sub-total",
            amount: { currency: "USD", value: "55.00" },
        },
        {
            label: "Sales Tax",
            amount: { currency: "USD", value: "5.00" },
            type: "tax"
        },
    ],
    total: {
        label: "Total due",
        /* Il totale è $60.00 perché dobbiamo aggiungere i costi di
   spedizione di $5.00.*/
        amount: { currency: "USD", value: "60.00" },
    },
};
```

Opzioni di spedizione
Qui vediamo un esempio di come aggiungere due opzioni di spedizione ai *details* [13].

```
const shippingOptions = [
    {
        id: "standard",
        label: "Ground Shipping (2 days)",
        amount: { currency: "USD", value: "5.00" },
        selected: true,
    },
    {
        id: "drone",
        label: " Drone Express (2 hours)",
        amount: { currency: "USD", value: "25.00" }
    },
];
Object.assign(details, { shippingOptions });
```

Modifiche condizionali alla richiesta di pagamento
Qui vediamo come aggiungere una tassa di elaborazione per l'utilizzo di una carta di credito. Si noti che richiede il ricalcolo del totale.

```
// La carta di credito comporta una commissione di $3.00
const creditCardFee = {
    label: "Credit card processing fee",
    amount: { currency: "USD", value: "3.00" },
};
//  I modifiers si applicano quando l'utente sceglie di pagare con una
   carta di credito
const modifiers = [
    {
        additionalDisplayItems: [creditCardFee],
        supportedMethods: "basic-card",
        total:
        {
            label: "Total due",
            amount: { currency: "USD", value: "68.00" },
        },
        data:
        {
            supportedTypes: "credit",
        },
    },
];
Object.assign(details, { modifiers });
```

PaymentOptions
Options contiene informazioni che lo sviluppatore ha bisogno dall'utente per eseguire il pagamento.

```
const options = {
    requestPayerEmail: false,
    requestPayerName: true,
    requestPayerPhone: false,
    requestShipping: true,
}
```

PaymentRequest
Dopo aver raccolto tutti i bit di informazioni prerequisite, ora possiamo costruirne uno *PaymentRequest* e richiedere che il browser lo presenti all'utente.

```
async function doPaymentRequest() {
    try{
        const request = new PaymentRequest(methodData, details, options)
    ;
        request.onshippingaddresschange = ev => ev.updateWith(details);
        request.onshippingoptionchange = ev => ev.updateWith(details);
        const response = await request.show();
        await validateResponse(response);
    } catch (err){
            // AbortError, SecurityError
            console.error(err);
        }
}
async function validateResponse(response){
    try {
        if (await checkAllValuesAreGood(response)){
            await response.complete("success");
        }
        else{
            await response.complete("fail");
        }
    } catch (err){
            // Qualcosa è andato storto
            await response.complete("fail");
        }
}
doPaymentRequest();
```

3.2.2 Implementazione su pagina d'esempio

Costruttore

L'oggetto *PaymentRequest* è costruito passando i seguenti parametri:

- **methodData:** una serie di identificativi del metodo di pagamento e tutti i dati pertinenti. Un identificativo del metodo di pagamento è una stringa che identifica un metodo di pagamento supportato;

- **details:** contiene le informazioni sulla transazione, come gli elementi pubblicitari in un ordine;

- **options:** contiene informazioni aggiuntive che il Wallet potrebbe dover raccogliere.

Nel seguente esempio stiamo consentendo agli utenti di pagare con qualsiasi carta di debito o di credito appartenente alle reti Visa, MasterCard o Amex. L'oggetto *details* contiene l'importo totale parziale, l'imposta sulle vendite e il totale dovuto; questi dettagli verranno mostrati all'utente nel portafoglio. Bisogna tenere presente che l'API non aggiunge elementi o calcola l'imposta sulle vendite, spetta al commerciante fornire le informazioni corrette. In questo esempio, stiamo vendendo un bene fisico, quindi chiediamo l'indirizzo di spedizione del cliente.

```
var methodData = [
    {
        supportedMethods: ['basic-card'],
        data: {
            supportedNetworks: ['visa', 'mastercard', 'amex'],
            supportedTypes: ['credit']
        }
    }
];
var details =  {
    displayItems: [
        {
            label: "Sub-total",
            amount: { currency: "USD", value : "100.00" } // US$100.00
        },
        {
            label: "Sales Tax",
            amount: { currency: "USD", value : "9.00" } // US$9.00
        }
    ],
    total:  {
        label: "Total due",
        amount: { currency: "USD", value : "109.00" } // US$109.00
    }
};
var options = {
    requestShipping: true
};
var payment = new PaymentRequest(methodData, details, options);
```

Visualizzazione dell'interfaccia utente, elaborazione del pagamento e visualizzazione dei risultati

Una volta creato l'oggetto *PaymentRequest* è possibile attivare il browser per visualizzare il wallet con *request.show()* [1].

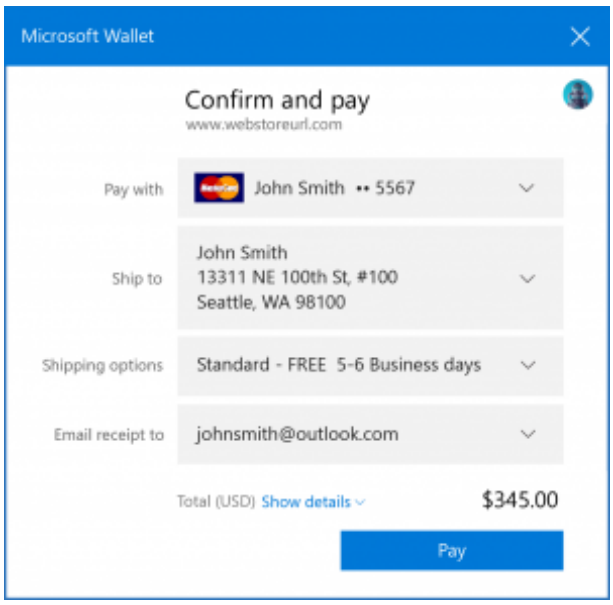

Figura 3.2: Wallet dopo la chiamata request.show()

I clienti possono selezionare le informazioni di pagamento, l'indirizzo di spedizione e altri campi appropriati e cliccare su Paga quando è pronto. A questo punto, gli utenti dovranno verificare la loro identità: in caso di esito positivo verrà soddisfatta la promise *request.show()* e verranno restituite al sito Web tutte le informazioni che il cliente ha fornito. Per il metodo di pagamento con carta di base l'oggetto risultante conterrà il nome del titolare della carta, il numero della carta, il mese di scadenza e altri campi pertinenti. Il commerciante può quindi utilizzare queste informazioni per elaborare la transazione sul back-end. Dopo che la risposta è tornata dal server, è possibile utilizzare *result.complete('success')* per visualizzare la schermata di successo o *result.complete('fail')* per indicare una transazione fallita.

```
1  //Quando la promessa è soddisfatta passa i risultati al server per l'
       elaborazione
2  payment.show().then(result => {
3      return process(result).then(response => {
4          if (response.status === 200) {
5              //La transazione ha avuto successo
6              return result.complete('success');
7          } else {
8                  //La transazione ha fallito
9                  return result.complete('fail');
10             }
11         }).catch((err) => {
12         console.error('User rejected request', err.message)
13     });
14 });
```

Ed ecco i wallet in caso di successo e di fallimento.

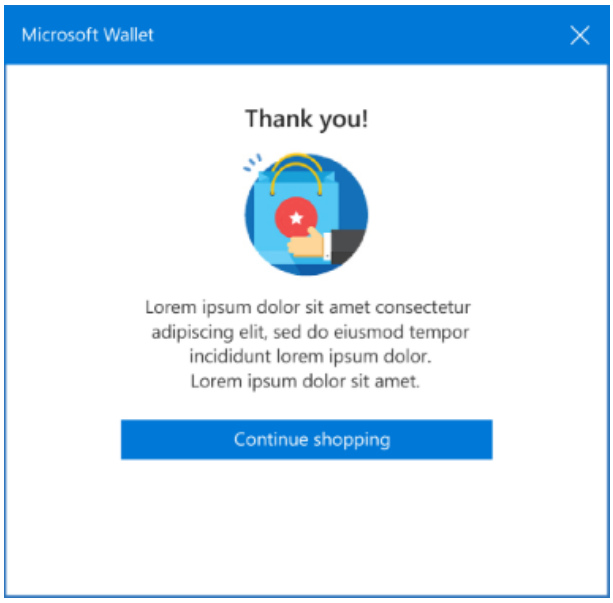

Figura 3.3: Wallet in caso di successo

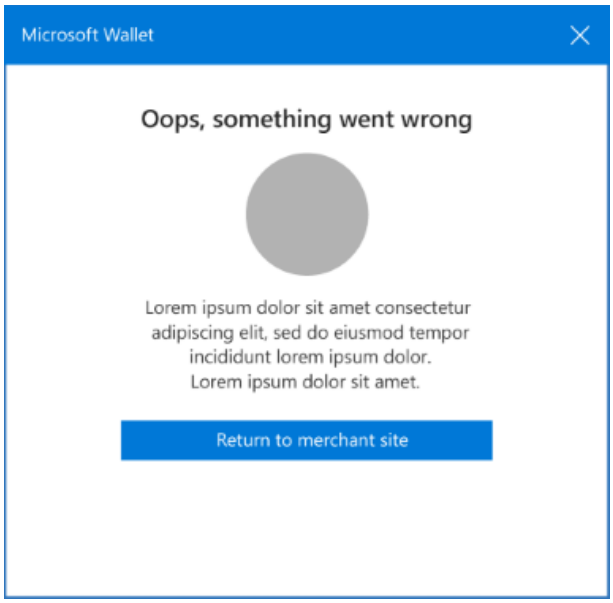

Figura 3.4: Wallet in caso di fail

3.3 Rischi e sicurezza [14]

La PaymentRequest API cerca di garantire la sicurezza dei dati sensibili nei seguenti modi:

- Per garantire che gli utenti non condividano inavvertitamente informazioni sensibili con una certa origine, l'API richiede che il proprio metodo *show()* venga attivato solo tramite la volontà da parte dell'utente, ad esempio con un clic. Inoltre, per evitare un'esperienza utente confusa, questa specifica limita la visualizzazione di una finestra alla volta del metodo *show()* così da non rischiare di inserire dati in più pagine, le quali potrebbero risultare untrusted. Se infatti si prova ad aprire più di una finestra dell'API accadrà la seguente situazione:

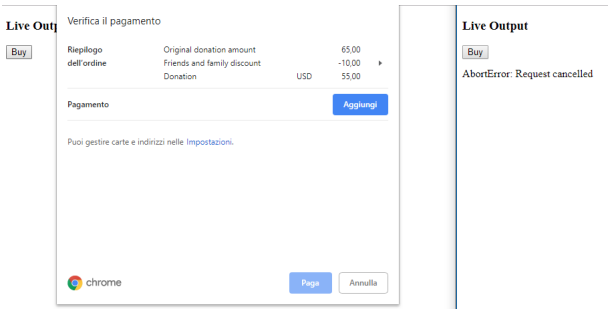

Figura 3.5: Metodo show() richiamato più volte

- La PaymentRequest API è disponibile solamente su HTTPS, quindi lavora solamente in contesti sicuri per aumentare la sicurezza dell'inserimento delle informazioni.
- I commercianti possono delegare la memorizzazione delle informazioni ai fornitori di servizi di pagamento o al browser, senza avere l'onere di memorizzare le informazioni sensibili dell'utente. Inoltre, ovviamente, le informazioni dell'utente non devono essere condivise con uno sviluppatore senza l'autorizzazione dell'utente.

Purtroppo un possibile rischio potrebbe essere quello di essere vittima di phishing nel caso fossimo in una pagina untrusted, ovvero viene aperta una pagina identica a quella in cui siamo e tutte le credenziali che inseriamo vengono mandate all'attaccante invece che al sito dove pensiamo di essere. Un altro possibile rischio invece potrebbe essere quello di essere vittima di un attacco XSS, ovvero un tipo di attacco nel quale degli script dannosi vengono inseriti lato server all'interno di siti affidabili, dove l'utente è ignaro di questo (questo tipo di attacco verrà spiegato nel prossimo paragrafo).
Chiaramente esistono diversi modi per mitigare questi rischi, ad esempio attraverso soluzioni di salvaguardia tecnologica e/o attraverso la consapevolezza dell'utente di tali rischi.

3.3.1 Esempio di un possibile attacco [15]

Descrizione dell'attacco
Supponiamo di essere all'interno di un sito affidabile, come Amazon o Facebook, e supponiamo di essere in una pagina di pagamento, come può essere il carrello su Amazon o il pagamento per un'inserzione su Facebook. Se un attaccante riesce a inserire degli script all'interno di queste pagine riuscirà a ribare le informazioni di pagamento di un utente. Quando l'utente cliccherà sulla voce "paga", in una delle due piattaforme le quali sono considerate affidabili, verrà eseguito lo script malevolo, invece di quello trusted del sito. L'utente inserirà tutte le

informazioni necessarie al pagamento ma queste verranno mandate e/o visualizzate dall'attaccante. Come mostrato nell'esempio in figura 14 in questo modo tutte le informazioni possono essere visualizzate dall'attaccante. Alla richiesta di avvio di un processo di pagamento, ovvero dopo la chiamata al metodo *show()*, viene creato l'oggetto PaymentRequest nel quale vengono inseriti tutti i dati che l'utente inserisce.

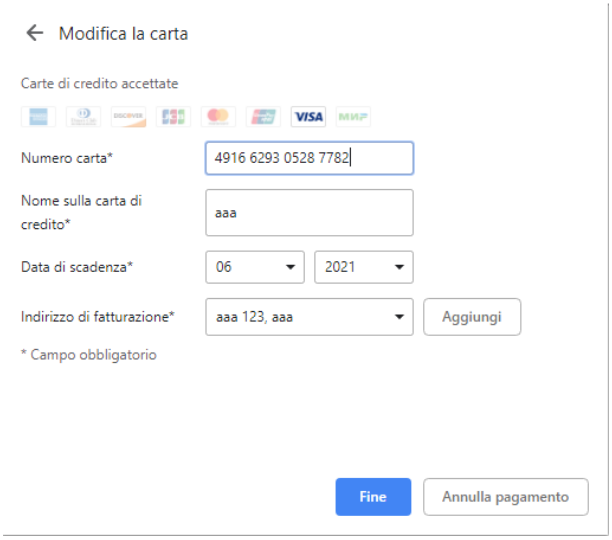

Figura 3.6: Inserimento informazioni dell'utente

Una volta che l'utente ha approvato una richiesta di pagamento viene restituito un *PaymentResponse* per approvare tale richiesta.

Figura 3.7: Inserimento CVV

L'oggetto *PaymentRequest*, creato in precedenza e contenente i dati dell'utente, passa però attraverso il DOM, e quindi può essere intercettato e può essere visualizzato ogni campo inserito dall'utente. Ad esempio nel campo *details.cardNumber* si può leggere il numero di carta di credito, mentre in *details.cardSecurityCode* si può leggere il codice CVV; entrambi sono mostrati nell'esempio nella figura seguente.

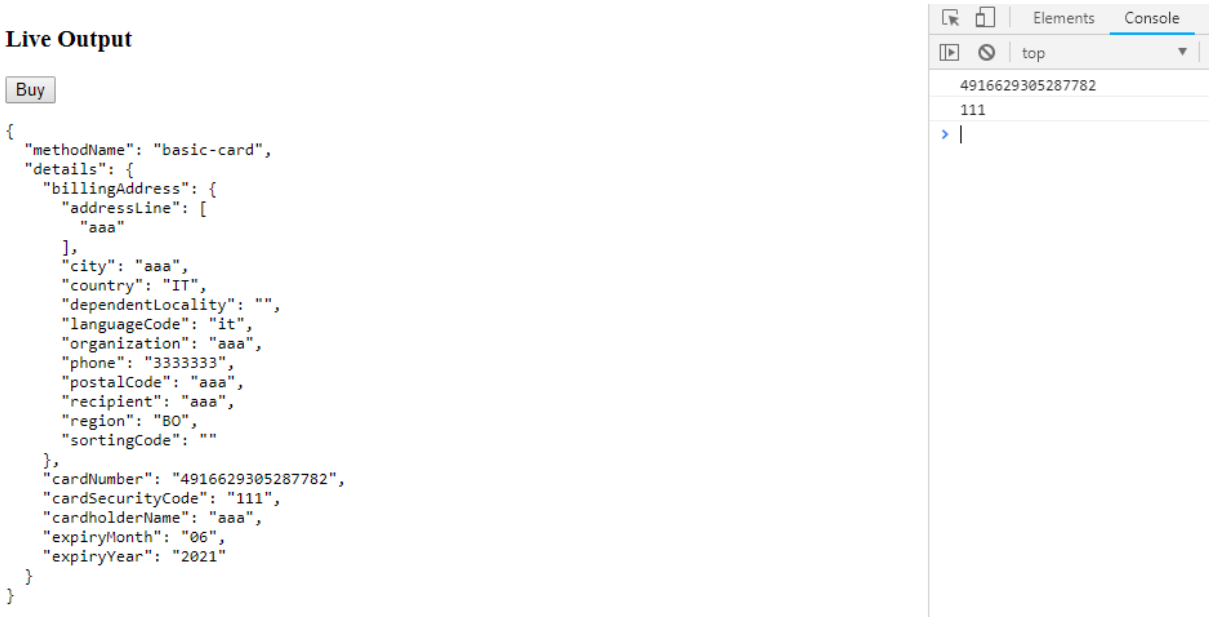

Figura 3.8: Informazioni rubate

3.4 Compatibilità web

3.4.1 Desktop

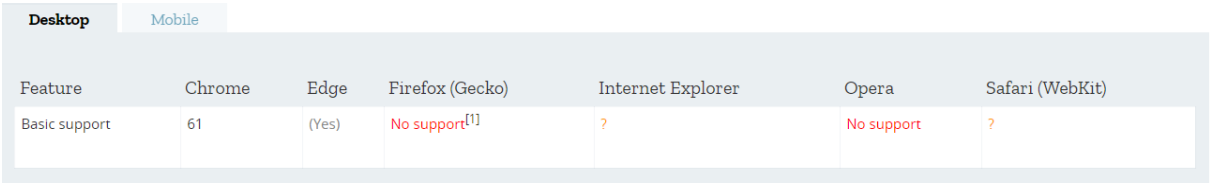

Figura 3.9: Compatibilità desktop

3.4.2 Mobile

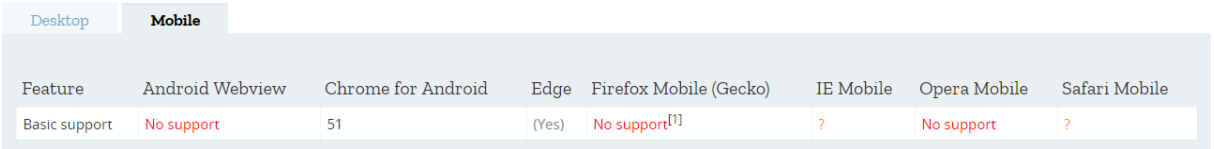

Figura 3.10: Compatibilità mobile

3.5 Conclusioni

La PaymentRequest API è uno strumento per migliorare l'esperienza utente sul Web offrendo ai clienti un'esperienza di acquisto più piacevole ma, come ogni strumento informatico, ha delle vulnerabilità. Come nel paragrafo precedente, anche qui abbiamo analizzato le specifiche e i problemi di privacy che tale API può avere e, come prima, identificheremo i vari threat model. Per quanto riguarda il migliorare l'esperienza utente abbiamo visto come questa API semplifichi notevolmente le esperienze di pagamento memorizzando le informazioni e riutilizzandole per completare in maniera più veloce le transazioni online. Considerando il secondo threat model, dove un utente usa l'API a fin di bene non sapendo però dell'esistenza del rischio, possiamo nominare due situazioni scritte nel capitolo 3.4: il phishing e l'attacco XSS. Prendendo in esame l'ultimo scenario, ovvero un attaccante che usa le API intenzionalmente in modo malevolo, si rimanda al paragrafo 3.4.1, dove vengono rubate le informazioni sensibili di un utente usando l'API ritenuta affidabile; questo di solito avviene nei siti non sicuri o, come detto prima, attraverso il phishing o gli attacchi XSS. I comportamenti per mitigare i rischi sono gli stessi della geolocation API, ovvero usare software di salvaguardia e/o avere consapevolezza di tali rischi all'interno del web.

CAPITOLO 4

SERVICE WORKER

4.1 Overview

Un Service Worker [16] [17] [18] è uno script Javascript che utilizza le *Promises* [19] per poter eseguire operazioni in modalità asincrona nel browser, avviate in background separato dalla pagina; pertanto non possono modificarne gli elementi del DOM come i normali script ma può comunicare con essi mediante "messaggi". Un Service Worker si trova tra la nostra applicazione Web e la rete e, come un server proxy, può intercettare tutte le richieste a pagine web e file statici e rispondere secondo politiche che siamo noi stessi a decidere. I Service Worker sono pensati per consentire la creazione di esperienze offline efficaci [20], intercettare le richieste di rete e intraprendere azioni appropriate in base al fatto che la rete sia disponibile o meno e aggiornare le risorse che risiedono sul server, oltre a consentire l'accesso alle notifiche push e alle API di sincronizzazione in background. È il browser che in qualsiasi momento deciderà se il Service Worker dovrebbe essere o meno in esecuzione così da risparmiare risorse, specialmente sui dispositivi mobili. Per questo può essere che se non facciamo alcuna richiesta HTTP per un certo periodo di tempo o non riceviamo alcuna notifica per un po' è possibile che il browser spenga il Service Worker. Se attiviamo una richiesta HTTP che deve essere gestita dal Service Worker il browser la attiverà di nuovo, nel caso in cui non fosse ancora in esecuzione.

4.1.1 Impostare i Service Worker [21]

Molte funzionalità dei Service Worker oggi sono abilitate di default, ma nel caso non lo fossero bisogna abilitarle nel browser:

- Firefox: su `about:config` impostare *dom.serviceWorkers.enabled* su true, riavvia il browser;
- Chrome : su `chrome://flags` accendere *experimental-web-platform-features*, riavvia browser;
- Opera : su `opera://flags` attivare *Support for Service Worker*, riavvia il browser;
- Microsoft Edge : su `about:flags` spuntare *Enable Service Workers*, riavvia il browser.

4.1.2 Architettura di base

Per quanto riguarda i Service Worker generalmente vengono eseguiti questi passaggi per l'impostazione di base:

- L'URL del Service Worker viene recuperato e registrato tramite *serviceWorkerContainer.register()*;
- In caso di esito positivo il Service Worker viene eseguito in un *ServiceWorkerGlobalScope*, ovvero un tipo speciale di ServiceContext che scappa dal thread di esecuzione dello script principale senza accesso DOM. Il Service Worker ora è pronto per elaborare gli eventi;
- L'installazione del Service Worker viene tentata quando si accede successivamente alle pagine: un evento di installazione è sempre il primo inviato a un Service Worker;
- Quando il Service Worker è considerato installato il passo successivo è l'attivazione, quindi quando il Service Worker è installato riceve un evento di attivazione.

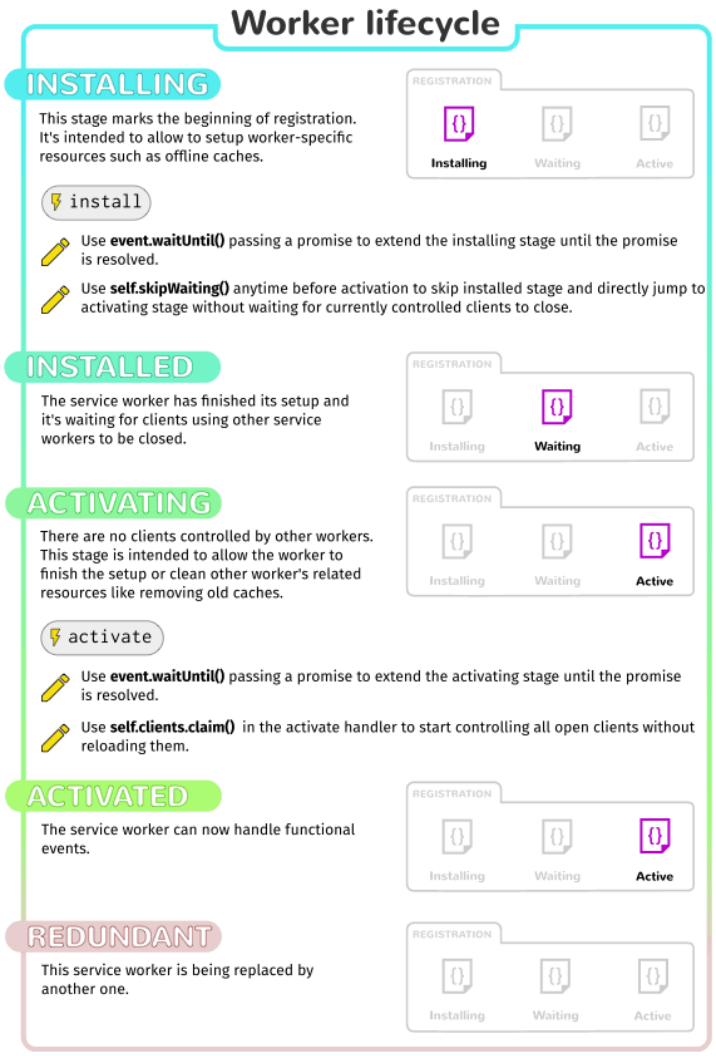

Figura 4.1: Ciclo di vita del Service Worker

4.1.3 Casi d'uso

I Service Worker sono destinati anche ad altri usi:

- Sincronizzazione dei dati in background: viene avviato un Service Worker anche quando nessun utente si trova sul sito, quindi le cache possono essere aggiornate, ecc;
- Risposta a richieste di risorse da altre origini;
- Caching;
- Miglioramenti delle prestazioni, ad esempio prelettura delle risorse che l'utente probabilmente avrà bisogno nel prossimo futuro.
- Reagire per inviare messaggi: si può avviare un Service Worker per inviare agli utenti un messaggio per comunicare loro che sono disponibili nuovi contenuti;
- Reagendo ad orari e date particolari.

4.2 Ciclo di vita di un Service Worker

Il ciclo di vita di un Service Worker [22] [23] [24] è composto da quattro fasi:

- **Registrazione**: il Service Worker viene scaricato dal browser, analizzato ed eseguito;
- **Installazione**: il Service Worker viene installato;
- **Attivazione**: il Service Worker è pronto ed è in grado di poter controllare gli eventi generati dal client;
- **Fetch**: evento generato dal client. Il Service Worker è in grado di intercettare le richieste e rispondere secondo le opportune strategie di caching.

4.2.1 Registrazione

Come prima cosa bisogna comunicare al browser l'esistenza di un ServiceWorker all'interno del sito web. Un Service Worker viene prima registrato utilizzando il metodo *ServiceWorker.register()* e per farlo basta inserire su tutte le pagine del sito uno script come il seguente:

```
1 if ('serviceWorker' in navigator) {
2   // Path che contiene il Service Worker
3     navigator.serviceWorker.register('/service-worker.js').then(function
    (registration) {
4       console.log('Service worker installato correttamente, ecco lo
    scope:', registration.scope);
5     }).catch(function(error) {
6       console.log('Installazione service worker fallita:', error);
7     });
8 }
```

Il codice inizia controllando il supporto da parte del browser verificando la presenza di *navigator.serviceWorker*. Se supportato, il Service Worker viene registrato per mezzo di *navigator.serviceWorker.register* che restituisce un oggetto *Promise* il quale si risolve con successo a registrazione avvenuta correttamente. *service-worker.js* è il file Javascript residente nella root del sito web e che contiene il codice del Service Worker, il cui codice è:

```
// Evento install
self.addEventListener('install', event => {
    // Codice da eseguire su installazione
    console.log("Service Worker Installato");
});
// Evento activate
self.addEventListener('activate', event => {
    // Codice da eseguire su attivazione
    console.log("Service Worker Attivo");
});
// Evento fetch
self.addEventListener('fetch', event => {
    // Codice da eseguire su fetch di risorse
    console.log("Richiesta URL: "+event.request.url);
});
```

In questo modo viene registrato un Service Worker il quale viene semplicemente installato e ad ogni richiesta stampa in console un messaggio con la URL che il browser tenta di scaricare dal server web. Per controllare il caricamento di un Service Worker il codice di questo deve essere eseguito al di fuori delle normali pagine.

Possono esserci diversi motivi per cui il Service Worker non si registra:

- Non si sta eseguendo l'applicazione tramite HTTPS;
- Il path del Service Worker non è scritto correttamente: deve essere scritto in relazione all'origine, non alla directory radice dell'app;
- Il Service Worker a cui ci si riferisce ha un'origine diversa da quella dell'app.

4.2.2 Installazione

Il Service Worker viene scaricato immediatamente quando un utente accede per la prima volta a un sito, o una pagina, controllata dal Service Worker, e sarà poi scaricato periodicamente ogni tot periodo di tempo.

L'installazione viene tentata quando il file nuovo che è stato scaricato risulta diverso da un Service Worker esistente, o risulta essere diverso dal primo Service Worker rilevato per quella pagina/sito. Se è la prima volta che un Service Worker viene reso disponibile viene tentata l'installazione e, dopo un'installazione corretta, viene attivato. Se è disponibile un Service Worker esistente la nuova versione viene installata in background, ma non ancora attivata; si attiva solo quando non ci sono più pagine caricate che stanno ancora utilizzando il vecchio Service Worker. Non appena non ci sono più pagine da caricare il nuovo Service Worker si attiva.

Conseguentemente all'installazione viene richiamato l'evento *install*: tale evento consente di effettuare il precaching, ovvero inserire in cache pagine e file statici del sito web prima di intercettarne le richieste. Per farlo occorre utilizzare gli oggetti *Promise* e cache come segue:

```
'use strict';
// Array di configurazione del service worker
var config = {
    version: 'versionesw1::',
    // Risorse da inserire in cache immediatamente - Precaching
    staticCacheItems: [
        '/wp-includes/js/jquery/jquery.js', '/wp-content/themes/miotema/logo.png', '/wp-content/themes/miotema/fonts/opensans.woff', '/wp-content/themes/miotema/fonts/fontawesome-webfont.woff2',
    ],
};
// Funzione che restituisce una stringa da utilizzare come chiave per la cache
function cacheName (key, opts) {
    return `${opts.version}${key}`;
}
// Evento install
self.addEventListener('install', event => { event.waitUntil(
    // Inserisco in cache le URL configurate in config.staticCacheItems
    caches.open(cacheName('static', config) ).then(cache => cache.addAll(config.staticCacheItems)).then(() => self.skipWaiting()));
    /*self.skipWaiting() evita l'attesa, il che significa che il service worker si attivera immediatamente non appena conclusa l'installazione*/
    console.log("Service Worker Installato");
});
```

Se si decidesse di aggiungere/eliminare nuove risorse da inserire in cache bisognerà avere l'accortezza di cambiare il nome della versione del Service Worker ed eliminare dalla cache le risorse già presenti. Una cosa molto importante da sapere è che le risorse da inserire in cache in fase di precaching devono esistere realmente sul server web altrimenti il Service Worker genererà un errore fatale e l'installazione non andrà a buon fine. Il metodo *skipWaiting()* consente al Service Worker di passare allo stato di attivazione ad installazione conclusa e quindi essere subito operativo.

4.2.3 Attivazione

Una volta installato il Service Worker passa nello stato di attivazione. Se la pagina al momento è controllata da un altro Service Worker quello attuale passa in uno stato di attesa per poi diventare operativo al prossimo caricamento di pagina quando il vecchio Service Worker viene sostituito. Questo per essere sicuri che solo un Service Worker (o una sola versione di Service Worker) per volta possa essere eseguito nello stesso contesto. A Service Worker attivato viene richiamato l'evento activate, ovvero l'evento per svuotare la cache obsoleta dell'eventuale precedente versione di Service Worker. Dopodiché il Service Worker sarà in grado di effettuare il fetching di risorse o di restare in attesa di altri eventi. Di default il nuovo Service Worker diventa operativo al refresh della pagina o dopo aver richiamato il metodo *clients.claim()*; fino a quel momento le eventuali richieste non saranno intercettate.

4.2.4 Fetch

Grazie all'evento fetch il Service Worker potrà agire da proxy tra l'applicazione web e la rete. Il Service Worker intercetterà ogni richiesta HTTP del browser e sarà in grado di rispondere a quest'ultimo prendendo la risorsa dalla cache piuttosto che scaricarla dalla rete. Grazie

all'evento fetch il Service Worker diventa un vero e proprio strumento per migliorare le performance di caricamento di un sito web.

4.2.5 Aggiornare il Service Worker

Se il Service Worker è già stato installato ma una nuova versione è disponibile per l'aggiornamento o il caricamento della pagina, la nuova versione viene installata in background ma non sarà ancora attivata. Si attiva solo quando non ci sono più pagine caricate che stanno ancora utilizzando il vecchio servizio. Non appena non ci sono più pagine di questo tipo ancora caricate, il nuovo Service Worker si attiverà.
Si dovrà aggiornare il listener install di eventi nel nuovo Service Worker, similmente a questo:

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v2').then(function(cache) {
      return cache.addAll([
        '/sw-test/',
        '/sw-test/index.html',
        '/sw-test/style.css',
        '/sw-test/app.js',
        '/sw-test/image-list.js',
        ...
      ]);
    })
  );
});
```

Mentre accade questo è ancora la versione precedente (v1) quella responsabile per i recuperi, mentre la nuova versione (v2) si sta installando in background. Quando nessuna pagina sta utilizzando la versione corrente, il nuovo operatore si attiva e diventa responsabile dei recuperi.

4.2.6 Disintallare il Service Worker

Rimuovere/disinstallare un Service Worker è un'operazione semplice. È possibile eseguirla manualmente dal proprio browser oppure inserendo un semplice script al posto di quello di registrazione del Service Worker:

```
navigator.serviceWorker.getRegistrations().then(function(registrations)
    {
    for(let registration of registrations) {
        registration.unregister()
    }
})
```

Naturalmente è necessario che la pagina contenente il codice di disinstallazione venga visitata dal browser, oppure è possibile rimuovere il Service Worker manualmente tramite DevTools.

4.3 Caching

Diverse sono le strategie che possono essere adottate per migliorare le performance di un sito web mediante i Service Worker. A seconda del sito e del contesto è possibile adottare una strategia piuttosto che l'altra:

- **Network first**
  Questa strategia mira ad avere un contenuto sempre fresco scaricandolo dalla rete, fornendo la copia in cache solo in caso di problemi di connettività, ad esempio in caso di connessione offline;

- **Cache first**
  Questa strategia verifica se la risorsa è disponibile in cache: se così fosse viene restituita la copia in cache, in caso contrario la risorsa viene scaricata dalla rete;

- **Network only**
  Viene simulato il normale comportamento del browser, ovvero scaricare le risorse direttamente dalla rete;

- **Cache only**
  In questo caso il Service Worker risponde solo con elementi conservati in cache: in caso di miss la risposta restituita al browser simulerà l'errore di connessione;

- **Fastest**
  Questa strategia mira a fornire all'utente la risposta più veloce: il Service Worker avvia contemporaneamente una richiesta in cache ed una in rete e la prima che risponde verrà restituita all'utente. Questa soluzione può essere l'ideale per quei dispositivi con vecchi hard drive dove la lettura da disco può rivelarsi più lenta del fetch dalla rete, mentre per i dispositivi moderni è meglio utilizzare la strategia cache then network.

- **Cache then network**
  Questa strategia mira a fornire il contenuto dalla cache per una risposta molto rapida avviando una richiesta in rete scaricando una copia aggiornata della risorsa e sostituendola con quella in cache; la risorsa ricevuta dalla rete viene poi sostituita con quella presente sulla pagina.

È importante sottolineare che il Service Worker non utilizza cache a meno che non siamo noi a dirlo, quindi di default il comportamento nella fase di fetch delle risorse sarà quello nativo del browser.

4.4 Rischi e sicurezza [25][1]

Come già detto i Service Worker operano solo in contesti protetti, ma questo non vuol dire che l'ambiente sia sicuro al 100% in quanto un Service Worker ha la possibilità di importare script da qualsiasi altra origine tramite la chiamata a *importScripts*, aumentando la capacità di un attaccante XSS di inserire il proprio codice javascript all'interno della pagina, potendo cosi rubare informazioni dell'utente, ad esempio all'inserimento di username e password in una data pagina, e portarle fuori. La registrazione dei Service Worker specifica che essi devono essere eseguiti nella stessa origine dei loro chiamanti; il confronto dell'origine è una corrispondenza col prefisso più lungo degli URL serializzati compreso il percorso, quindi ad esempio *https://example.com* è differente da *https://example.evil.com*, quindi un attaccante può effettivamente registrare un Service Worker malevolo. Per mitigare questo rischio il browser richiede che l'URL di registrazione del Service Worker provenga dall'origine stessa; quindi per registrare un Service Worker malevolo attraverso un attacco XSS l'utente malintenzionato ha bisogno di ospitare i propri script sul server.
Con questo possibile scenario, dove una pagina ha una vulnerabilità XSS, potrebbe essere registrato un Service Worker il quale chiama *importScripts* per importare script malevoli da terze parti. In una situazione XSS del genere il limite della direttiva cache di 24 ore garantisce che un Service Worker malevolo o compromesso sopravviverà a un massimo di 24 ore, o meno in base a come è impostato il sito. Una possibile mitigazione del problema potrebbe essere accorciare la vita dei Service Worker, ovviamente in modo ragionevole altrimenti non sarebbero sfruttate le potenzialità.
Quindi un Service Worker potrebbe non essere usato per miglioare i tempi di risposta dell'applicazione o del sito tramite il caching, ma potrebbe anche essere usato per intercettare messaggi, modificarli (e in tal caso restituirli errati similmente a man-in-the-middle, come verrà mostrato nell'esempio al paragrafo seguente).

[1] `https://alf.nu/ServiceWorker`

4.4.1 Esempio di attacco

Supponiamo di avere questa pagina HTML che carica uno script per l'installazione di un Service Worker, pensando sia sicuro.

```
<html>
    <head></head>
    <body>
    <script type="text/javascript">
            navigator.serviceWorker.register('http://127.0.0.1/script.js
    ').then(function(registration){
                console.log(registration)
            });
        </script>
        <script src="./install.js"></script>
    </body>
</html>
```

Il file *script.js* sarà il seguente:

```
self.addEventListener('fetch', function (event) {
    event.respondWith(
        caches.match(event.request).then(function(res){
                if(res){  //Se c'è una cache usala
                    return res;
                }
                return requestBackend(event);
            })
    )
});

function requestBackend(event){
    var url = event.request.clone();
    console.log(url)
    if(url.url=='http://127.0.0.1/index.html'){
    //determina se le risorse devono essere dirottate
    return new Response("<script>alert(1)</script>", { headers: { '
    Content-Type': 'text/html' }})
    }
    return fetch(url).then(function(res){
        if(!res || res.status !== 200 || res.type !== 'basic'){
            return res;
        }
        var response = res.clone();
        caches.open('v1').then(function(cache){
            cache.put(event.request, response);
        });
        return res;
    })
}
```

Mentre *hack.js*, che intercetta ogni richiesta, sarà questo:

```
this.addEventListener ('fetch', function (event) {
    event.respondWith (new Response ("Intercepted!"));
});
```

Infine il file *install.js* che installerà il Service Worker malevolo sarà questo:

```
if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('/hack.js').then(function(
    registration) {
        console.log('ServiceWorker registration successful with scope: '
    , registration.scope);
    })
};
```

Da questo vediamo che, fino a che il Service Worker non sarà disinstallato e non verrà cancellata la cache, ogni richiesta che verrà fatta sarà intercettata: in questo esempio sarà mostrato il messaggio "Intercepted" nella pagina web ma ovviamente un Web Attacker potrebbe intercettare queste richieste e usarle come detto nel paragrafo precedente.

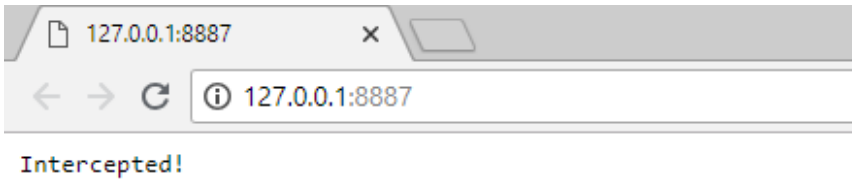

Figura 4.2: Intercepted

4.5 Compatibilità web

4.5.1 Desktop

Desktop | Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	40.0	16[2]	33.0 (33.0)[1]	No support	24	No support

Figura 4.3: Compatibilità web

4.5.2 Mobile

Desktop | **Mobile**

Feature	Android Webview	Chrome for Android	Firefox Mobile (Gecko)	Firefox OS	IE Phone	Opera Mobile	Safari Mobile
Basic support	No support	40.0	(Yes)	(Yes)	No support	(Yes)	No support

Figura 4.4: Compatibilità mobile

4.6 Conclusioni

Dopo questo lungo discorso si è capito che i Service Worker sono uno strumento potentissimo, e come ogni strumento possono essere usati in modo benevolo o malevolo. Se realizzati per fare caching possono rendere la navigazione del sito web o dell'applicazione molto veloce senza rendere necessarie modifiche al sito o all'applicazione per raggiungere tale scopo. Purtroppo, come si è visto negli ultimi paragrafi, sono anche uno potente strumento a scopo malevolo. Anche qui, come nei due capitoli precedenti, abbiamo analizzato le specifiche e i problemi di privacy che tale API può avere e, come fatto in precedenza, identificheremo i vari threat model. Per quanto riguarda il primo caso, le migliorie dell'esperienza utente riguardono certamente il caching e la velocità con il quale le informazioni vengono caricate molto più velocemente. Il secondo e il terzo threat model sono simili alla Payment Request API, ovvero si tratta di vulnerabilità XSS. Per mitigare i rischi vi sono le direttive scritte al paragrafo precedente, ovvero il fatto che il limite di tempo di un Service Worker nella cache è di 24 ore, o meno in base alle impostazioni; in alternativa, o in aggiunta, si possono comunque usare, come nel caso delle altre API, software di sicurezza per proteggersi dai rischi all'interno del web.

CAPITOLO 5

CONCLUSIONI

Questa tesi ha cercato di rispondere alla domanda: *"Come faccio ad individuare i rischi di queste APIs e come posso affrontarli?"*. A tal fine è stato preso un campione di tre APIs e sono state condotte delle ricerche e degli studi per rivelare le vulnerabilità di ognuna di esse, trovandone diverse in ognuna di quelle prese in considerazione.
Come già detto durante tutta la tesi tali vulnerabilità possono essere colmate, o comunque evitate; chiaramente sono avantaggiate a tale riguardo le persone sempre a contatto coi mezzi tecnlogici e sicuramente anche le nuove generazioni le quali sono cresciute in mezzo alla tecnologia. Questo è un fattore da tenere in considerazione dato che a mio parere alcuni rischi, come immettere i propri dati o consentire di rilevare la propria posizione in un sito non sicuro, sono prevalenti in certe fasce d'età. Questa considerazione si basa sulla mia esperienza personale e, per questo motivo, non è una dichiarazione generale e oggettiva. Il dato oggettivo sono i problemi legati alle diverse APIs che ho espresso nel corso della mia tesi.
Una ricerca potrebbe essere quella di realizzare uno studio sui problemi della sicurezza in internet nelle diverse fasce d'età, concentrandosi invece che sulle vulnerabilità delle APIs su come cercare di educare più o meno tutti ai rischi del Web.

Ringraziamenti

A conclusione di questo lavoro di tesi mi sento in dovere di porre i miei più sentiti ringraziamenti alle persone che ho avuto modo di conoscere in questo importante periodo della mia vita e che mi hanno aiutato a crescere dal punto di vista professionale ma soprattutto umano.
Un sincero ringraziamento al mio relatore Dott. Stefano Calzavara che, oltre alla sua vastissima e precisa conoscenza nel campo della sicurezza, mi ha saputo risolvere ogni dubbio, e sopratutto mi ha aiutato a non lasciare stare questo lavoro quando la rassegnazione stava prendendo il sopravvento.
Un grazie anche al mio correlatore Dott. Marco Squarcina che ha saputo darmi consigli fondamentali per riuscire ad andare avanti in alcuni punti della tesi che altrimenti non avrei saputo risolvere da solo.
Non possono mancare in questo elenco tutti i miei compagni di corso con i quali ho trascorso questi anni nelle aule dell'università tra risate, scene uniche e indimenticabili e terrore durante qualche esame, con i quali ho instaurato un rapporto di amicizia che spero duri nel tempo.
Una dose importante di ringraziamenti va a tutti quei miei amici che, dicendomi la giusta dose di parole nei momenti opportuni ma aiutandomi altrettante volte e soprattutto spendendo gran parte del loro tempo per darmi una mano, sono riuscite a farmi compiere questo percorso. Grazie davvero.
Dovrei scrivere un'altra tesi se scrivessi ad una ad una tutte le persone che, in diverse forme, mi hanno aiutato a superare tutti i problemi e i brutti periodi di questi anni universitari. Sicuramente uno dei motivi per cui sono arrivato fino a qua è stato anche grazie al continuo "Ce la fai, non mollare!" in risposta ai miei "Lascio stare tutto, non sono capace".
Per ultimi, ma non meno importanti, i miei genitori. Non so se troverò mai le parole giuste per ringraziarvi, però vorrei che questo traguardo ripagasse, per quanto possibile, tutti i sacrifici che hanno fatto per me, e che fosse un premio anche per loro. Un grande grazie a tutta la mia famiglia per esserci sempre, per sostenermi in ogni cosa io faccia, e sicuramente senza i vostri consigli e le vostre critiche non sarei la persona che sono oggi. Grazie per avermi fatto arrivare a questo punto di arrivo e contemporaneamente di partenza della mia vita.

ELENCO DELLE FIGURE

BIBLIOGRAFIA

[1] Mozilla Developer. Features restricted to secure contexts. https://developer.mozilla.org/en-US/docs/Web/API/Geolocation.

[2] Mozilla Developer. Payment request api, mozilla developer. https://developer.mozilla.org/en-US/docs/Web/API/Payment_Request_API.

[3] Mozilla Developer. Geolocation api. https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API.

[4] W3C. Geolocation api specification 2nd edition. 2018. https://www.w3.org/TR/geolocation-API/.

[5] Aurelio De Rosa. An introduction to the geolocation api. 2014. https://code.tutsplus.com/tutorials/an-introduction-to-the-geolocation-api--cms-20071.

[6] Nick Doty, Deirdre K. Mulligan, and Erik Wilde. Privacy issues of the w3c geolocation api. 2010. https://escholarship.org/uc/item/0rp834wf.

[7] Rafay Baloch. Html5 modern day attack and defense vector. http://www.xss-payloads.com/papers/HTML5AttackVectors.pdf.

[8] Mozilla Developer. Payment request api. 2018. https://w3c.github.io/payment-request/.

[9] Google Developers. Introduction to the payment request api. https://developers.google.com/web/ilt/pwa/introduction-to-the-payment-request-api.

[10] Microsoft. Simpler web payments: Introducing the payment request api. https://blogs.windows.com/msedgedev/2016/12/15/payment-request-api-edge/#UATsm3ejAT9oYtrj.97.

[11] Matt Gaunt. Deep dive into payment request api. https://developers.google.com/web/fundamentals/payments/merchant-guide/deep-dive-into-payment-request.

[12] Eiji Kitamura. Bringing easy and fast checkout with payment request api. https://developers.google.com/web/updates/2016/07/payment-request.

[13] Microsoft. Payment request api samples. https://developer.microsoft.com/en-us/microsoft-edge/testdrive/demos/paymentrequest/.

[14] W3C. Paymentrequest privacy and security considerations. https://w3c.github.io/payment-request/#privacy-and-security-considerations.

[15] Google Inc. Paymentrequest credit cards sample. 2018. https://googlechrome.github.io/samples/paymentrequest/credit-cards/.

[16] Google Partners. Tecnologie web avanzate: Service worker. https://support.google.com/partners/answer/7336697?hl=it.

[17] Speedy Wordpress. Guida completa ai service worker javascript. https://www.speedywordpress.it/guida-completa-ai-service-worker-javascript/.

[18] Matt Gaunt. Service workers: an introduction. https://developers.google.com/web/fundamentals/primers/service-workers/.

[19] Mozilla Developer. Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[20] Nicolas Bevacqua. Making a simple site work offline with serviceworker. 2015. https://css-tricks.com/serviceworker-for-offline/.

[21] Mozilla Developer. Setting up to play with service workers. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers.

[22] Mozilla Developer. Service worker concepts and usage. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[23] Angular University. Service workers - practical guided introduction. https://blog.angular-university.io/service-workers/.

[24] jakearchibald Github. Service worker explained. 2017. https://github.com/w3c/ServiceWorker/blob/master/explainer.md.

[25] W3C. Service workers nightly. 2018. https://w3c.github.io/ServiceWorker/#security-considerations.