前端的安全问题

同源策略

是一种约定，是浏览器最核心也最基本的安全功能，限制了来自不同源的document或者脚本，对当前document读取或设置某些属性

- 影响“源”的因素有：host（域名或者IP地址）、子域名、端口、协议
- 对浏览器来说，DOM、Cookie、XMLHttpRequest会受到同源策略的限制

不受同源策略的标签

`<script>`、`<img>`、`<iframe>`、`<link>`等标签都可以跨域加载资源，而不受同源策略的限制

- 这些带"src"属性的标签每次加载时，浏览器会发起一次GET请求
- 通过src属性加载的资源，浏览器限制了javascript的权限，使其不能读、写返回的内容

==========

1.xss攻击

xss攻击又叫做跨站脚本攻击，主要是用户输入或通过其他方式，向我们的代码中注入了一下其他的js，而我们又没有做任何防范，去执行了这段js。

可能用户会写一个死循环，将我们的页面给弄崩了，但是也有可能通过这种方式，来获取我们的cookie，从而回去登陆态等信息。

xss攻击从来源可分为反射型和存储型

反射型：

将xss代码通过url来注入在IE浏览器去访问该页面并在地址后面加上

```
index.html#<img src="404.html"
onerror="alert('xss')" />
```

这时页面一打开就会有个xss的弹窗，这就是最简单的反射型攻击

当然可能会觉得这样没有任何作用，但是修改一下代码

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>反射型</title>
```

```
</head>
<body>
<div id="test"></div>

<script>
  // 先向页面的cookie存储一个name=1的信息
    document.cookie = "name=1"
    var $test = document.querySelector('#test');;
    $test.innerHTML = window.location.hash
</script>
</body>
</html>
```

此时打开的地址修改为`index.html#<img src="404.html" onerror="alert(document.cookie)" />`这里就会发现弹窗内容为我们存取的cookie。

注意：1.这里必须用IE打开这个链接，因为chrome和safari等浏览器，会主动将url里的一下字符串进行encode，保证了一定的安全性。 2.为什么我们这里用img的onerror来注入脚本呢？而不是直接用script标签来执行，我们修改一下访问的地址 `index.html#<script>alert(document.cookie)</script>`**，这时会发现，页面并没有执行这段代码，但是这段代码已经注入到了#test标签中了。所以，一般通过img的onerror来注入是最有效的方法。**

存储型

将xss代码发送到了服务器，在前端请求数据时，将xss代码发送给了前端。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>存储型</title>
</head>
<body>
<div id="test"></div>

<script>
  // 先向页面的cookie存储一个name=1的信息
    document.cookie = "name=1"
    // 这里假设是请求了后台的接口 response是我们请求回来的数据
    var response = '<img src="404.html"
```

```
onerror="alert(document.cookie)"'

    var $test = document.querySelector('#test');;
    $test.innerHTML = response
</script>
</body>
</html>
```

这里最常见的情况就是一个富文本编辑器下，由用户输入了一串xss代码，存储在了服务器中，我们在展示用户输入内容时，没有做防范处理。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>富文本</title>
</head>
<body>
  <div id="test"></div>
  <textarea name="" id="" cols="30" rows="10"></textarea>
  <button onclick="submit()">提交</button>
</body>
</html>
<script>
function submit() {
  var $test = document.querySelector('#test');
    $test.innerHTML = document.querySelector('textarea').value
}
</script>
```

xss的防范手段：

1.encode

encode也分为html的encode和js的encode

html的encode： 就是将一些有特殊意义的字符串进行替换，比如：

& => &

" => "

' => '

< => <

> => >

通过这些字符的替换，之前我们输入的`<img src="null" onerror="alert()">`就被encode成了`<img src=&quot;null&quot; onerror=&quot;alert()&quot;>`

js的encode： 使用“\”对特殊字符进行转义，除数字字母之外，小于127的

字符编码使用16进制“\xHH”的方式进行编码，大于用unicode（非常严格模式）
用“\”对特殊字符进行转义,这个可能比较好理解，因为将一下，比如', "这些字符转译为',"就可以使得js变为一个字符串，而不是一个可执行的js代码了，那为什么还需要进行16进制转换和unicode转换呢？这样做是为了预防一下隐藏字符，比如换行符可能会对js代码进行换行

```
//使用“\”对特殊字符进行转义，除数字字母之外，小于127使用16进制“\xHH”的方式进行编码，大于用unicode（非常严格模式）。
var JavaScriptEncode = function(str){

    var hex=new Array('0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f');

    function changeTo16Hex(charCode){
        return "\\x" + charCode.charCodeAt(0).toString(16);
    }

    function encodeCharx(original) {

        var found = true;
        var thecharchar = original.charAt(0);
        var thechar = original.charCodeAt(0);
        switch(thecharchar) {
            case '\n': return "\\n"; break; //newline
            case '\r': return "\\r"; break; //Carriage return
            case '\'': return "\\'"; break;
            case '"': return "\\\""; break;
            case '\&': return "\\&"; break;
            case '\\': return "\\\\"; break;
            case '\t': return "\\t"; break;
            case '\b': return "\\b"; break;
            case '\f': return "\\f"; break;
            case '/': return "\\x2F"; break;
            case '<': return "\\x3C"; break;
            case '>': return "\\x3E"; break;
            default:
                found=false;
                break;
        }
        if(!found){
            if(thechar > 47 && thechar < 58){ //数字
```

```
                return original;
            }

            if(thechar > 64 && thechar < 91){ //大写字母
                return original;
            }

            if(thechar > 96 && thechar < 123){ //小写字母
                return original;
            }

            if(thechar>127) { //大于127用unicode
                var c = thechar;
                var a4 = c%16;
                c = Math.floor(c/16);
                var a3 = c%16;
                c = Math.floor(c/16);
                var a2 = c%16;
                c = Math.floor(c/16);
                var a1 = c%16;
                return "\
\u"+hex[a1]+hex[a2]+hex[a3]+hex[a4]+"";
            }
            else {
                return changeTo16Hex(original);
            }

        }
    }

    var preescape = str;
    var escaped = "";
    var i=0;
    for(i=0; i < preescape.length; i++){
        escaped = escaped + encodeCharx(preescape.charAt(i));
    }
    return escaped;
}
```

2.对于富文本的防范：filter

因为富文本是比较特殊的，在富文本中输入标签，我们需要展示出来，所以我们不能用之前的html的encode方法来执行。所以我们就得用一个叫白名单过滤的方式来防范。原理就是：首先列举一下比较合法的标签，称

为白名单，这些标签是不会对页面进行攻击的。之后对用户输入的内容进行白名单过滤。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>富文本</title>
</head>
<body>
  <div id="test"></div>
  <textarea name="" id="" cols="30" rows="10"></textarea>
  <button onclick="submit()">提交</button>
</body>
</html>
<!-- xss_filter.js是一个白名单过滤库 -->
<script src="./xss_filter.js"></script>
<script>
function submit() {
  var $test = document.querySelector('#test');
    $test.innerHTML =
filterXSS(document.querySelector('textarea').value)
}
</script>
```

此时用户如果提交`<img src="null" onerror="alert()">`就会被过滤成`<img src="">`

这样就有效的防范了用户输入的xss代码

注意： 当遇到想后台提交数据的情况，我们应该在用户提交的时候就进行过滤和encode呢？还是展示的时候处理呢？ 我们应当考虑到提交代码后会有个多端展示的问题，可能我们web端需要进行这些处理，但是移动端展示的时候就不需要这些处理，所以我们应当在展示的时候进行处理，而不是录入的时候处理

2、append的利用上一小节我们防住了script标签的左右尖括号，但聪明的黑客们还是想出了好办法去破解，我们知道，直接给innerHTML赋值一段js，是无法被执行的。比如，

```
1  $('div').innerHTML = '<script>alert("okok");</script>';
```

复制代码

但是，jQuery的append可以做到，究其原因，就是因为jquery会在将

append元素变为fragment的时候，找到其中的script标签，再使用eval执行一遍。jquery的append使用的方式也是innerHTML。而innerHTML是会将unicode码转换为字符实体的。

利用这两种知识结合，我们可以得出，网站使用append进行dom操作，如果是append我们可以决定的字段，那么我们可以将左右尖括号，使用unicode码伪装起来，就像这样--"\u003cscript\u003ealert('xss');"。接下来转义的时候，伪装成\u003的<会被漏掉，append的时候，则会被重新调用。虽然进行了转义，注入的代码还是会再次被执行

XSS的防御

1、HttpOnly

浏览器禁止页面的Javascript访问带有HttpOnly属性的cookie。（实质解决的是：XSS后的cookie劫持攻击）如今已成为一种“标准”的做法

不同语言给cookie添加HttpOnly的方式不同，比如

- JavaEE：`response.setHeader("Set-Cookie","cookiename=value; Path=/;Domain=domainvalue;Max-Age=seconds;HTTPOnly");`
- PHP4：`header("Set-Cookie:hidden=value;httpOnly");`
- PHP5：`setcookie("abc","test",NULL,NULL,NULL,NULL,TRUE); //true为HttpOnly属性`

2、输入检查（XSS Filter）

- 原理：让一些基于特殊字符的攻击失效。（常见的Web漏洞如XSS、SQLInjection等，都要求攻击者构造一些特殊字符）
- 输入检查的逻辑，必须放在服务器端代码中实现。目前*Web*开发的普遍做法，是同时哎客户端*Javascript*中和服务端代码中实现相同的输入检查。客户端的输入检查可以阻挡大部分误操作的正常用户，节约服务器资源。

3、输出检查

在变量输出到HTML页面时，使用编码或转义的方式来防御XSS攻击

- 针对HTML代码的编码方式：HtmlEncode
- PHP：htmlentities()和htmlspecialchars()两个函数
- Javascript：JavascriptEncode（需要使用“”对特殊字符进行转义，

同时要求输出的变量必须在引号内部）
- 在URL的path（路径）或者search（参数）中输出，使用URLEncode

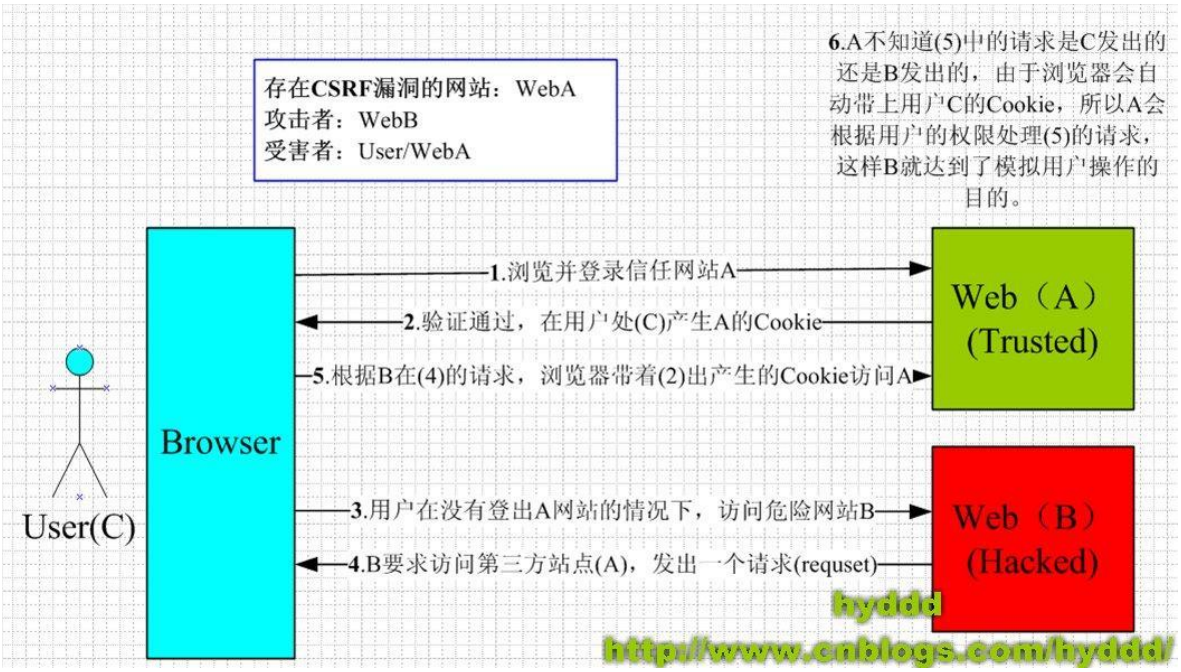
防御DOM Based XSS
- **DOM Based XSS的形成：**（举个例子）
- **实质：**从Javascript中输出数据到HTML页面里
- **这个例子的解决方案：**做一次HtmlEncode

防御方法：分语境使用不同的编码函数

2.CSRF– 跨站伪造请求

攻击者首先在自己的域构造一个页面：`http://www.a.com/csrf.html`，其内容为`<img src="http://blog.sohu.com/manage/entry.do?m=deleted&id=156714243" />`
使用了一个img标签，其地址指向了删除Id为156714243的博客文章
然后攻击者诱使目标用户，也就是博客主人访问这个页面
用户进去看到一张无法显示的图片，这时自己的那篇博客文章已经被删除了

CSRF就是利用你所在网站的登录的状态，悄悄提交各种信息， 是一种比xss还要恶劣很多的攻击。因为CSRF可以在我们不知情的情况下，利用我们登陆的账号信息，去模拟我们的行为，去执行一下操作，也就是所谓的钓鱼。比如我们在登陆某个论坛，但这个网站是个钓鱼网站，我们利用邮箱或者qq登陆后，它就可以拿到我们的登陆态，session和cookie信息。然后利用这些信息去模拟一个另外网站的请求，比如转账的请求。



csrf.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>csrf</title>
</head>
<body>
<iframe id="" name="csrf-form"></iframe>
<form target="csrf-form" method="post" action="http://
127.0.0.1:3001/csrf">
    <input name="name" value="1111">
    <input type="submit" value="提交">
</form>
</body>
</html>
```

我们点击进入了一个csrf这个页面里，我们以为我们只是在csrf中点击提交了1111这个信息，其实这个网站悄悄的把这些信息提交到了本地的csrf上了，而不是我们当前浏览的csrf.html中

server.js

```
const http = require('http');
const fs = require('fs');
const proxy = http.createServer((req, res) => {

    if(req.method == 'POST'){
```

```
        req.on('data' , (data)=>{
            console.log('referer :' ,   req.headers.referer);
            console.log('data :' ,   data.toString() , '
cookies:' , req.headers.cookie);
        });

        req.on('end' , (data)=>{
            res.writeHead(200, { 'Content-Type': 'text/
html' });

            res.end('');
        })
    } else {
        res.setHeader('Set-Cookie', ['login=1']);

        res.end('');
    }

}).listen(3001);
```

在命令行中输入`node server.js`
这是一个简单的服务，端口为3001，如果我们直接本地登陆`localhost:3001`会给我们本地注入一个cookie为login=1的登陆态此时我们在访问csrf.html，在点击提交按钮的时候，会发现会把这个登陆态也提交上去。这就是一个典型的钓鱼网站，

防范措施

1. 提交 method=Post 判断refererHTTP请求中有一个referer的报文头，用来指明当前流量的来源参考页。如果我们用post就可以将页面的referer带入，从而进行判断请求的来源是不是安全的网站。但是referer在本地起的服务中是没有的，直接请求页面也不会有。这就是为什么我们要用Post请求方式。直接请求页面，因为post请求是肯定会带入referer，但get有可能不会带referer。
2. 利用Token
   Token简单来说就是由后端生成的一个唯一的登陆态，并传给前端保存在前端，每次前端请求时都会携带着Token，后端会先去解析这个Token，看看是不是后台给我们的，已经是否登陆超时，如果校验通过了，才会同意接口请求

需要注意的点：

1. Token需要足够随机，必须用足够安全的随机数生成算法
2. Token应该为用户和服务器所共同持有，不能被第三方知晓
3. Token可以放在用户的Session或者浏览器的Cookie中
4. 尽量把Token放在表单中，把敏感操作由GET改为POST，以form表单的形式提交，可以避免Token泄露（比如一个页面：`http://host/path/manage?username=abc&token=[random]`，在此页面用户需要在这个页面提交表单或者单击“删除”按钮，才能完成删除操作，在这种场景下，如果这个页面包含了一张攻击者能指定地址的图片`<img src="http://evil.com/notexist" />`，则这个页面地址会作为HTTP请求的Refer发送到evil.com的服务器上，从而导致Token泄露）

1、验证码

CSRF攻击过程中，用户在不知情的情况下构造了网络请求，添加验证码后，强制用户必须与应用进行交互

- 优点：简洁而有效
- 缺点：网站不能给所有的操作都加上验证码

2、Referer Check

利用HTTP头中的Referer判断请求来源是否合法

Referer首部包含了当前请求页面的来源页面的地址

- 优点：简单易操作（只需要在最后给所有安全敏感的请求统一添加一个拦截器来检查Referer的值就行）
- 缺点：服务器并非什么时候都能取到Referer
  1. 很多出于保护用户隐私的考虑，限制了Referer的发送。
  2. 比如从HTTPS跳转到HTTP，出于安全的考虑，浏览器不会发送Referer
  3.

3、点击劫持（ClickJacking）

什么是点击劫持

点击劫持是一种视觉上的欺骗手段。攻击者使用一个透明的、不可见的iframe，覆盖在一个网页上，然后诱使用户在网页上进行操作，此时用户将在不知情的情况下点击透明的iframe页面。通过调整iframe页面的位置，可以诱使用户恰好点击在iframe页面的一些功能性按钮上。

防御点击劫持：X-Frame-Options

X-Frame-Options HTTP响应头是用来给浏览器指示允许一个页面能否在`<frame>`、`<iframe>`、`<object>`中展现的标记

有三个可选的值

1. DENY：浏览器会拒绝当前页面加载任何frame页面（即使是相同域名的页面也不允许）
2. SAMEORIGIN：允许加载frame页面，但是frame页面的地址只能为同源域名下的页面
3. ALLOW-FROM：可以加载指定来源的frame页面（可以定义frame页面的地址）

四、控制台注入代码

不知道各位看官有没有注意到天猫官网控制台的警告信息，如图4.1所示，这是为什么呢？因为有的黑客会诱骗用户去往控制台里面粘贴东西（欺负小白用户不懂代码），比如可以在朋友圈贴个什么文章，说:"只要访问天猫，按下F12并且粘贴以下内容，则可以获得xx元礼品"之类的，那么有的用户真的会去操作，并且自己隐私被暴露了也不知道。
天猫这种做法，也是在警告用户不要这么做，看来天猫的前端安全做的也是很到位的。不过，这种攻击毕竟是少数，所以各位看官看一眼就行，如果真的发现有的用户会被这样攻击的话，记得想起天猫的这种解决方案

什么是SQL注入攻击
攻击者在HTTP请求中注入恶意的SQL代码，服务器使用参数构建数据库SQL命令时，恶意SQL被一起构造，并在数据库中执行。
用户登录，输入用户名 lianggzone，密码 ‘ or ‘1’=’1 ，如果此时使用参数构造的方式，就会出现
select * from user where name = ‘lianggzone’ and password = ‘’ or ‘1’=‘1’
不管用户名和密码是什么内容，使查询出来的用户列表不为空。如何防范SQL注入攻击使用预编译的PrepareStatement是必须的，但是一般我们会从两个方面同时入手。
Web端
1）有效性检验。
2）限制字符串输入的长度。
服务端
1）不用拼接SQL字符串。
2）使用预编译的PrepareStatement。
3）有效性检验。(为什么服务端还要做有效性检验？第一准则，外部都是

不可信的，防止攻击者绕过Web端请求）
4）过滤SQL需要的参数中的特殊字符。比如单引号、双引号。

DDos 攻击
客户端向服务端发送请求链接数据包，服务端向客户端发送确认数据包，客户端不向服务端发送确认数据包，服务器一直等待来自客户端的确认
没有彻底根治的办法，除非不使用TCP
DDos 预防：
1）限制同时打开SYN半链接的数目
2）缩短SYN半链接的Time out 时间
3）关闭不必要的服务
————————————————

==========

一、前端漏洞

1、XSS攻击

核心：恶意脚本注入
描述：攻击者通过在目标网站上注入恶意脚本，使之在用户的浏览器上运行。利用这些恶意脚本，攻击者可获取用户的敏感信息如 Cookie、SessionID 等，进而危害数据安全。

2、CSRF攻击

核心：利用用户身份伪造请求
描述：利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，冒充用户对被攻击的网站发送执行某项操作的请求

3、HTTP劫持

核心：广告、弹框html注入
描述：当我们访问页面的时候，运营商在页面的HTML代码中，插入弹窗、广告等HTML代码，来获取相应的利益

4、界面操作劫持

核心：视觉欺骗
描述：界面操作劫持是一种基于视觉欺骗的劫持攻击。通过在页面上覆盖一个iframe + opacity:0的页面，让用户误点击

5、错误的内容推断

核心：js伪装成图片文件

描述：攻击者将含有JavaScript的脚本文件伪装成图片文件（修改后缀等）。该文件逃过了文件类型校验，在服务器里存储了下来。接下来，受害者在访问这段评论的时候，浏览器请求这个伪装成图片的JavaScript脚本并执行

6、不安全的第三方依赖包

核心：第三方漏洞

描述：框架及第三方依赖的安全漏洞

7、HTTPS降级HTTP

核心：拦截首次http通信

描述：问题的本质在于浏览器发出去第一次请求就被攻击者拦截了下来并做了修改，根本不给浏览器和服务器进行HTTPS通信的机会。大致过程如下，用户在浏览器里输入URL的时候往往不是从https://开始的，而是直接从域名开始输入，随后浏览器向服务器发起HTTP通信，然而由于攻击者的存在，它把服务器端返回的跳转到HTTPS页面的响应拦截了，并且代替客户端和服务器端进行后续的通信

8、本地存储数据泄露

核心：敏感、机密数据

描述：前端存储敏感、机密信息易被泄露

9、缺失静态资源完整性校验

核心：CDN资源劫持

描述：存储在CDN中的静态资源，攻击者劫持了CDN，或者对CDN中的资源进行了污染

10、文件上传漏洞

核心：文件类型限制

描述：文件后缀及文件内容没有严格限制

11、文件下载漏洞

核心：文件类型、目录限制

描述：下载敏感文件、下载目录