

常见六大Web安全攻防解析

前言

在互联网时代，数据安全与个人隐私受到了前所未有的挑战，各种新奇的攻击技术层出不穷。如何才能更好地保护我们的数据？本文主要侧重于分析几种常见的攻击的类型以及防御的方法。

想阅读更多优质原创文章请猛戳[GitHub博客](#)

一、XSS

XSS (Cross-Site Scripting)，跨站脚本攻击，因为缩写和 CSS重叠，所以只能叫 XSS。跨站脚本攻击是指通过存在安全漏洞的Web网站注册用户的浏览器内运行非法的HTML标签或JavaScript进行的一种攻击。

跨站脚本攻击有可能造成以下影响：

- 利用虚假输入表单骗取用户个人信息。
- 利用脚本窃取用户的Cookie值，被害者在不知情的情况下，帮助攻击者发送恶意请求。
- 显示伪造的文章或图片。

XSS 的原理是恶意攻击者往 Web 页面里插入恶意可执行网页脚本代码，当用户浏览该页之时，嵌入其中 Web 里面的脚本代码会被执行，从而可以达到攻击者盗取用户信息或其他侵犯用户安全隐私的目的。

XSS 的攻击方式千变万化，但还是可以大致细分为几种类型。

1.非持久型 XSS（反射型 XSS）

非持久型 XSS 漏洞，一般是通过给别人发送**带有恶意脚本代码参数的 URL**，当 URL 地址被打开时，特有的恶意代码参数被 HTML 解析、执行。



举一个例子，比如页面中包含有以下代码：

```
<select>
  <script>
    document.write(''
      + '<option value=1>'
      + location.href.substring(location.href.indexOf('default=') + 8)
      + '</option>'
    );
    document.write('<option value=2>English</option>');
  </script>
</select>
```

攻击者可以直接通过 URL (类似：[https://xxx.com/xxx?default=<script>alert\(document.cookie\)</script>](https://xxx.com/xxx?default=<script>alert(document.cookie)</script>)) 注入可执行的脚本代码。不过一些浏览器如Chrome其内置了一些XSS过滤器，可以防止大部分反射型XSS攻击。

非持久型 XSS 漏洞攻击有以下几点特征：

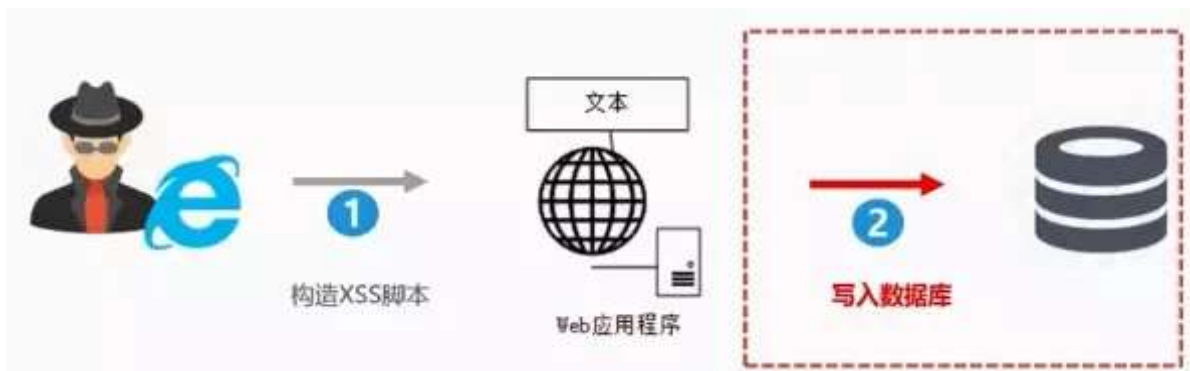
- 即时性，不经过服务器存储，直接通过 HTTP 的 GET 和 POST 请求就能完成一次攻击，拿到用户隐私数据。
- 攻击者需要诱骗点击,必须要通过用户点击链接才能发起
- 反馈率低，所以较难发现和响应修复
- 盗取用户敏感保密信息

为了防止出现非持久型 XSS 漏洞，需要确保这么几件事情：

- Web 页面渲染的所有内容或者渲染的数据都必须来自于服务端。
- 尽量不要从 `URL` , `document.referrer` , `document.forms` 等这种 DOM API 中获取数据直接渲染。
- 尽量不要使用 `eval` , `new Function()` , `document.write()` , `document.writeln()` , `window.setInterval()` , `window.setTimeout()` , `innerHTML` , `document.createElement()` 等可执行字符串的方法。
- 如果做不到以上几点，也必须对涉及 DOM 渲染的方法传入的字符串参数做 escape 转义。
- 前端渲染的时候对任何的字段都需要做 escape 转义编码。

2.持久型 XSS（存储型 XSS）

持久型 XSS 漏洞，一般存在于 Form 表单提交等交互功能，如文章留言，提交文本信息等，黑客利用的 XSS 漏洞，将内容经正常功能提交进入数据库持久保存，当前端页面获得后端从数据库中读出的注入代码时，恰好将其渲染执行。



举个例子，对于评论功能来说，就得防范持久型 XSS 攻击，因为我可以在评论中输入以下内容



主要注入页面方式和非持久型 XSS 漏洞类似，只不过持久型的不是来源于 URL，referer，forms 等，而是来源于**后端从数据库中读出来的数据**。持久型 XSS 攻击不需要诱骗点击，黑客只需要在提交表单的地方完成注入即可，但是这种 XSS 攻击的成本相对还是很高。

攻击成功需要同时满足以下几个条件：

- POST 请求提交表单后端没做转义直接入库。
- 后端从数据库中取出数据没做转义直接输出给前端。
- 前端拿到后端数据没做转义直接渲染成 DOM。

持久型 XSS 有以下几个特点：

- 持久性，植入在数据库中
- 盗取用户敏感私密信息
- 危害面广

3.如何防御

对于 XSS 攻击来说，通常有两种方式可以用来防御。

1) CSP

CSP 本质上就是建立白名单，开发者明确告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截是由浏览器自己实现的。我们可以通过这种方式来尽量减少 XSS 攻击。

通常可以通过两种方式来开启 CSP：

- 设置 HTTP Header 中的 Content-Security-Policy
- 设置 meta 标签的方式

这里以设置 HTTP Header 来举例：

- 只允许加载本站资源

```
Content-Security-Policy: default-src 'self'
```

- 只允许加载 HTTPS 协议图片

```
Content-Security-Policy: img-src https://*
```

- 允许加载任何来源框架

```
Content-Security-Policy: child-src 'none'
```

如需了解更多属性，请查看[Content-Security-Policy文档](#)

对于这种方式来说，只要开发者配置了正确的规则，那么即使网站存在漏洞，攻击者也不能执行它的攻击代码，并且 CSP 的兼容性也不错。

2) 转义字符

用户的输入永远不可信任的，最普遍的做法就是转义输入输出的内容，对于引号、尖括号、斜杠进行转义

```
function escape(str) {  
    str = str.replace(/&/g, '&amp;')  
    str = str.replace(/</g, '&lt;')  
    str = str.replace(/>/g, '&gt;')  
    str = str.replace(/"/g, '&quot;')  
    str = str.replace(/'/g, '&#39;')  
}
```

```
str = str.replace(/`/g, '&#96;')
str = str.replace(/\\/g, '&#x2F;')
return str
}
```

但是对于显示富文本来说，显然不能通过上面的办法来转义所有字符，因为这样会把需要的格式也过滤掉。对于这种情况，通常采用白名单过滤的办法，当然也可以通过黑名单过滤，但是考虑到需要过滤的标签和标签属性实在太多，更加推荐使用白名单的方式。

```
const xss = require('xss')
let html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>')
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;
console.log(html)
```

以上示例使用了 js-xss 来实现，可以看到在输出中保留了 h1 标签且过滤了 script 标签。

3) HttpOnly Cookie。

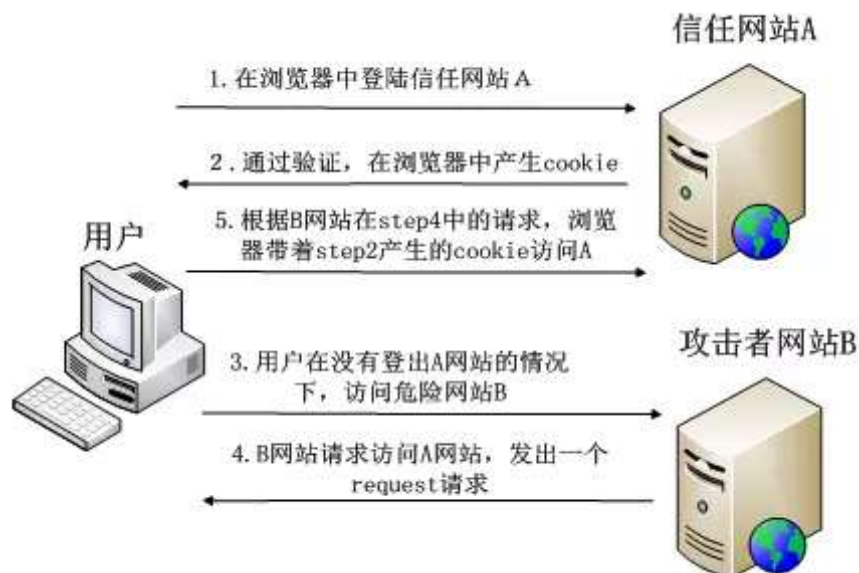
这是预防XSS攻击窃取用户cookie最有效的防御手段。Web应用程序在设置cookie时，将其属性设为 HttpOnly，就可以避免该网页的cookie被客户端恶意JavaScript窃取，保护用户cookie信息。

二、CSRF

CSRF(Cross Site Request Forgery)，即跨站请求伪造，是一种常见的Web攻击，它利用用户已登录的身份，在用户毫不知情的情况下，以用户的名义完成非法操作。

1.CSRF攻击的原理

下面先介绍一下CSRF攻击的原理：



完成 CSRF 攻击必须要有三个条件:

- 用户已经登录了站点 A, 并在本地记录了 cookie
- 在用户没有登出站点 A 的情况下 (也就是 cookie 生效的情况下), 访问了恶意攻击者提供的引诱危险站点 B (B 站点要求访问站点 A)。
- 站点 A 没有做任何 CSRF 防御

我们来看一个例子: 当我们登入转账页面后, 突然眼前一亮惊现"XXX隐私照片, 不看后悔一辈子"的链接, 耐不住内心躁动, 立马点击了该危险的网站 (页面代码如下图所示), 但当这页面一加载, 便会执行 `submitForm` 这个方法提交转账请求, 从而将10块转给黑客。

```
自动提交表单 转账地址 转账信息 提交表单

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>xxx隐私照片, 不看后悔一辈子</title>
<style>
.tip { width: 200px; margin: 20px auto; font-size: 20px; }
</style>
</head>
<body onload="submitForm();" >
  <div class="tip">加载中, 请稍候...</div>
  <form id="transferForm"
    action="http://127.0.0.1:8088/demo/csrf/transfer.php"
    method="post">
    <input type="hidden" name="toUser" value="黑客" />
    <input type="hidden" name="amount" value="10" />
  </form>
</body>
<script>
  function submitForm() {
    document.getElementById("transferForm").submit();
  }
</script>
</html>
```

2.如何防御

防范 CSRF 攻击可以遵循以下几种规则：

- Get 请求不对数据进行修改
- 不让第三方网站访问到用户 Cookie
- 阻止第三方网站请求接口
- 请求时附带验证信息，比如验证码或者 Token

1) SameSite

可以对 Cookie 设置 SameSite 属性。该属性表示 Cookie 不随着跨域请求发送，可以很大程度减少 CSRF 的攻击，但是该属性目前并不是所有浏览器都兼容。

2) Referer Check

HTTP Referer是header的一部分，当浏览器向web服务器发送请求时，一般会带上Referer信息告诉服务器是从哪个页面链接过来的，服务器籍此可以获得一些信息用于处理。可以通过检查请求的来源来防御CSRF攻击。正常请求的referer具有一定规律，如在提交表单的referer必定是在该页面发起的请求。所以**通过检查http包头referer的值是不是这个页面，来判断是不是CSRF攻击。**

但在某些情况下如从https跳转到http，浏览器处于安全考虑，不会发送referer，服务器就无法进行check了。若与该网站同域的其他网站有XSS漏洞，那么攻击者可以在其他网站注入恶意脚本，受害者进入了此类同域的网址，也会遭受攻击。出于以上原因，无法完全依赖Referer Check作为防御CSRF的主要手段。但是可以通过Referer Check来监控CSRF攻击的发生。

3) Anti CSRF Token

目前比较完善的解决方案是加入Anti-CSRF-Token。即发送请求时在HTTP 请求中以参数的形式加入一个随机产生的token，并在服务器建立一个拦截器来验证这个token。服务器读取浏览器当前域cookie中这个token值，会进行校验该请求当中的token和cookie当中的token值是否都存在且相等，才认为这是合法的请求。否则认为这次请求是违法的，拒绝该次服务。

这种方法相比Referer检查要安全很多，token可以在用户登陆后产生并放于session或cookie中，然后在每次请求时服务器把token从session或cookie中拿出，与本次请求中的token 进行比对。由于token的存在，攻击者无法再构造出一个完整的URL实施CSRF攻击。但在处理多个页面共存问题时，当某个页面消耗掉token后，其他页面的表单保存的还是被消耗掉的那个token，其他页面的表单提交时会出现token错误。

4) 验证码

应用程序和用户进行交互过程中，特别是账户交易这种核心步骤，强制用户输入验证码，才能完成最终请求。在通常情况下，验证码能够很好地遏制CSRF攻击。**但增加验证码降低了用户的体验，网站不能给所有的操作都加上验证码。**所以只能将验证码作为一种辅助手段，在关键业务点设置验证码。

三、点击劫持

点击劫持是一种视觉欺骗的攻击手段。攻击者将需要攻击的网站通过 iframe 嵌套的方式嵌入自己的网页中，并将 iframe 设置为透明，在页面中透出一个按钮诱导用户点击。

1. 特点

- 隐蔽性较高，骗取用户操作
- "UI-覆盖攻击"
- 利用iframe或者其它标签的属性

2. 点击劫持的原理

用户在登陆 A 网站的系统后，被攻击者诱惑打开第三方网站，而第三方网站通过 iframe 引入了 A 网站的页面内容，用户在第三方网站中点击某个按钮（被装饰的按钮），实际上是点击了 A 网站的按钮。接下来我们举个例子：我在优酷发布了很多视频，想让更多的人关注它，就可以通过点击劫持来实现

```
iframe {
width: 1440px;
height: 900px;
position: absolute;
top: -0px;
left: -0px;
z-index: 2;
-moz-opacity: 0;
opacity: 0;
filter: alpha(opacity=0);
}
button {
position: absolute;
top: 270px;
left: 1150px;
z-index: 1;
width: 90px;
height:40px;
```



```
}
</style>
.....
<button>点击脱衣</button>

<iframe src="http://i.youku.com/u/UMjA0NTg4Njcy" scrolling="no"></iframe>
```



从上图可知，攻击者通过图片作为页面背景，隐藏了用户操作的真实界面，当你按耐不住好奇点击按钮以后，真正的点击的其实是隐藏的那个页面的订阅按钮，然后就会在你不知情的情况下订阅了。



3. 如何防御

1) X-FRAME-OPTIONS

X-FRAME-OPTIONS 是一个 HTTP 响应头，在现代浏览器有一个很好的支持。这个 HTTP 响应头 就是为了防御用 iframe 嵌套的点击劫持攻击。

该响应头有三个值可选，分别是

- DENY，表示页面不允许通过 iframe 的方式展示
- SAMEORIGIN，表示页面可以在相同域名下通过 iframe 的方式展示
- ALLOW-FROM，表示页面可以在指定来源的 iframe 中展示

2) JavaScript 防御

对于某些远古浏览器来说，并不能支持上面的这种方式，那我们只有通过 JS 的方式来防御点击劫持了。

```
<head>
  <style id="click-jack">
    html {
      display: none !important;
    }
  </style>
</head>
<body>
  <script>
    if (self == top) {
      var style = document.getElementById('click-jack')
      document.body.removeChild(style)
    } else {
      top.location = self.location
    }
  </script>
</body>
```

以上代码的作用就是当通过 iframe 的方式加载页面时，攻击者的网页直接不显示所有内容了。

四、URL跳转漏洞

定义：借助未验证的URL跳转，将应用程序引导到不安全的第三方区域，从而导致的安全问题。

1.URL跳转漏洞原理

黑客利用URL跳转漏洞来诱导安全意识低的用户点击，导致用户信息泄露或者资金的流失。其原理是黑客构建恶意链接(链接需要进行伪装,尽可能迷惑),发在QQ群或者是浏览量多的贴吧/论坛中。安全意识低的用户点击后,经过服务器或者浏览器解析后，跳到恶意的网站中。



恶意链接需要进行伪装,经常的做法是熟悉的链接后面加上一个恶意的网址，这样才迷惑用户。

诸如伪装成像如下的网址，你是否能够识别出来是恶意网址呢？

```
http://gate.baidu.com/index?act=go&url=http://t.cn/RVTatrd  
http://qt.qq.com/safecheck.html?flag=1&url=http://t.cn/RVTatrd  
http://tieba.baidu.com/f/user/passport?jumpUrl=http://t.cn/RVTatrd
```

2.实现方式：

- Header头跳转
- Javascript跳转
- META标签跳转

这里我们举个Header头跳转实现方式：

```
<?php  
$url=$_GET['jumpto'];  
header("Location: $url");  
?>
```

```
http://www.wooyun.org/login.php?jumpto=http://www.evil.com
```

这里用户会认为 www.wooyun.org 都是可信的，但是点击上述链接将导致用户最终访问 www.evil.com 这个恶意网址。

3.如何防御

1)referer的限制

如果确定传递URL参数进入的来源，我们可以通过该方式实现安全限制，保证该URL的有效性，避免恶意用户自己生成跳转链接

2)加入有效性验证Token

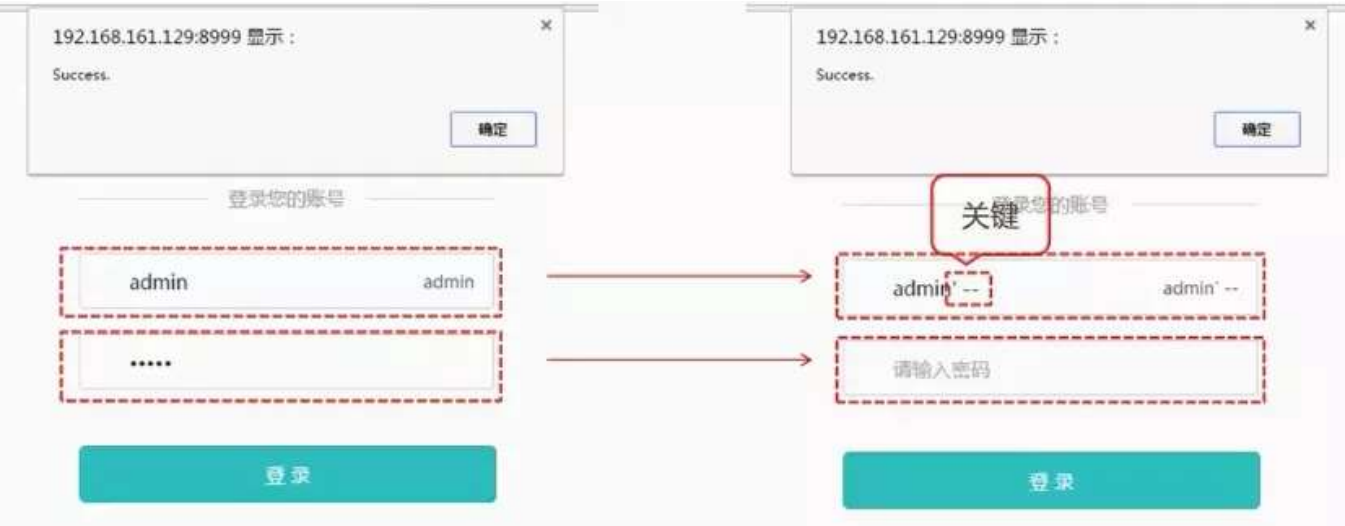
我们保证所有生成的链接都是来自于我们可信域的，通过在生成的链接里加入用户不可控的Token对生成的链接进行校验，可以避免用户生成自己的恶意链接从而被利用，但是如果功能本身要求比较开放，可能导致有一定的限制。

五、SQL注入

SQL注入是一种常见的Web安全漏洞，攻击者利用这个漏洞，可以访问或修改数据，或者利用潜在的数据库漏洞进行攻击。

1.SQL注入的原理

我们先举一个万能钥匙的例子来说明其原理：



```
<form action="/login" method="POST">
  <p>Username: <input type="text" name="username" /></p>
  <p>Password: <input type="password" name="password" /></p>
  <p><input type="submit" value="登陆" /></p>
</form>
```

后端的 SQL 语句可能是如下这样的：

```
let querySQL = `
  SELECT *
  FROM user
  WHERE username='${username}'
  AND psw='${password}'
```

```
`;  
// 接下来就是执行 sql 语句...
```

这是我们经常见到的登录页面，但如果有一个恶意攻击者输入的用户名是 `admin' --`，密码随意输入，就可以直接登入系统了。why! ----这就是SQL注入

我们之前预想的SQL 语句是:

```
SELECT * FROM user WHERE username='admin' AND psw='password'
```

但是恶意攻击者用奇怪用户名将你的 SQL 语句变成了如下形式:

```
SELECT * FROM user WHERE username='admin' --' AND psw='xxxx'
```

在 SQL 中, `' --` 是闭合和注释的意思, `--` 是注释后面的内容的意思, 所以查询语句就变成了:

```
SELECT * FROM user WHERE username='admin'
```

所谓的万能密码,本质上就是SQL注入的一种利用方式。

一次SQL注入的过程包括以下几个过程:

- 获取用户请求参数
- 拼接到代码当中
- SQL语句按照我们构造参数的语义执行成功

SQL注入的必备条件: 1.可以控制输入的数据 2.服务器要执行的代码拼接了控制的数据。



我们会发现SQL注入流程中与正常请求服务器类似，只是黑客控制了数据，构造了SQL查询，而正常的请求不会SQL查询这一步，**SQL注入的本质:数据和代码未分离，即数据当做了代码来执行。**

2.危害

- 获取数据库信息
 - 管理员后台用户名和密码
 - 获取其他数据库敏感信息：用户名、密码、手机号码、身份证、银行卡信息.....
 - 整个数据库：脱裤
- 获取服务器权限
- 植入Webshell，获取服务器后门
- 读取服务器敏感文件

3.如何防御

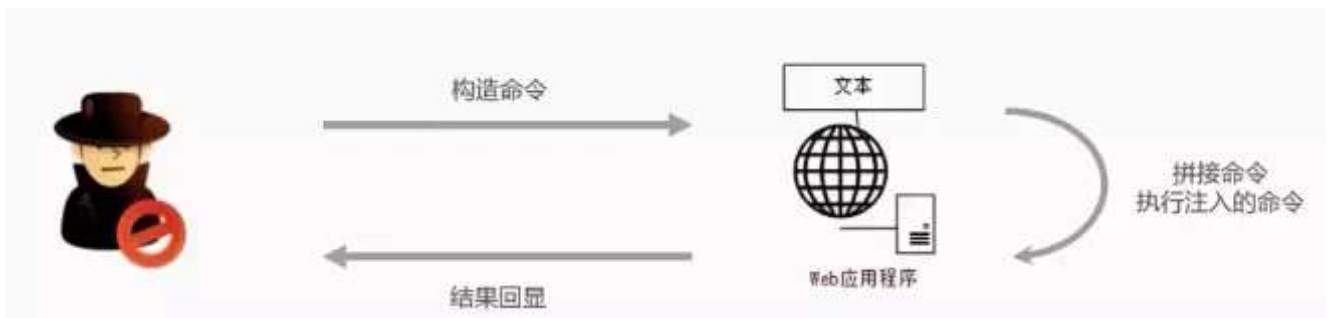
- **严格限制Web应用的数据库的操作权限**，给此用户提供仅仅能够满足其工作的最低权限，从而最大限度的减少注入攻击对数据库的危害
- **后端代码检查输入的数据是否符合预期**，严格限制变量的类型，例如使用正则表达式进行一些匹配处理。
- **对进入数据库的特殊字符（'，"，\，<，>，&，*，；等）进行转义处理，或编码转换**。基本上所有的后端语言都有对字符串进行转义处理的方法，比如 lodash 的 `lodash.escapehtmlchar` 库。
- **所有的查询语句建议使用数据库提供的参数化查询接口**，参数化的语句使用参数而不是将用户输入变量嵌入到 SQL 语句中，即不要直接拼接 SQL 语句。例如 Node.js 中的 `mysqljs` 库的 `query` 方法中的 `？` 占位参数。

六、OS命令注入攻击

OS命令注入和SQL注入差不多，只不过SQL注入是针对数据库的，而OS命令注入是针对操作系统的。OS命令注入攻击指通过Web应用，执行非法的操作系统命令达到攻击的目的。只要在能调用Shell函数的地方就有存在被攻击的风险。倘若调用Shell时存在疏漏，就可以执行插入的非法命令。

命令注入攻击可以向Shell发送命令，让Windows或Linux操作系统的命令行启动程序。也就是说，通过命令注入攻击可执行操作系统上安装着的各种程序。

1.原理



黑客构造命令提交给web应用程序，web应用程序提取黑客构造的命令，拼接到被执行的命令中，因黑客注入的命令打破了原有命令结构，导致web应用执行了额外的命令，最后web应用程序将执行的结果输出到响应页面中。

我们通过一个例子来说明其原理，假如要实现一个需求：用户提交一些内容到服务器，然后在服务器执行一些系统命令去返回一个结果给用户

```
// 以 Node.js 为例，假如在接口中需要从 github 下载用户指定的 repo
const exec = require('mz/child_process').exec;
let params = { /* 用户输入的参数 */ };
exec(`git clone ${params.repo} /some/path`);
```

如果 `params.repo` 传入的是 `https://github.com/admin/admin.github.io.git` 确实能从指定的 git repo 上下载到想要的代码。但是如果 `params.repo` 传入的是 `https://github.com/xx/xx.git && rm -rf /* &&` 恰好你的服务是用 root 权限起的就糟糕了。

2.如何防御

- 后端对前端提交内容进行规则限制（比如正则表达式）。
- 在调用系统命令前对所有传入参数进行命令行参数转义过滤。
- 不要直接拼接命令语句，借助一些工具做拼接、转义预处理，例如 Node.js 的 `shell-escape` npm 包

给大家推荐一个好用的BUG监控工具[Fundebug](#)，欢迎免费试用！



微信搜一搜

前端工匠

参考资料

- [常见Web 安全攻防总结](#)
- [前端面试之道](#)
- [图解Http](#)
- [Web安全知多少](#)
- [web安全之点击劫持\(clickjacking\)](#)
- [URL重定向/跳转漏洞](#)
- [网易web白帽子](#)

关注下面的标签，发现更多相似文章

JavaScript

浪里行舟 前端小白 @ 厦门
获得点赞 7,402 次 · 文章被阅读 142,516 次

安装掘金浏览器插件

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

评论

输入评论...

imicro

<script>alert(1)</script>

1月前

👍 1 💬 回复

采也车格

回复 imicro: 你想攻击掘金吗 😂

9天前

猿七 前端开发一枚~

写的还不错，受益匪浅。 😊 难得一篇好文。

1月前


👍 💬 回复

浪里行舟 (作者) 前端小白 @ 厦门

回复 猿七: 欢迎关注我的公众号：前端工匠，分享更多优质文章
1月前

猿七 前端开发一枚~
回复 猿七: 好的。
1月前



加载更多

OBKoro1 前端 @ 上海
<script>alert(1)</script>
1月前   回复

OBKoro1 前端 @ 上海
回复 OBKoro1: 掘金做了防护啊 你们干嘛呢 🤔
1月前

2shou1496797143475
<script>alert('Hshddh')</script>
1月前   回复

JimminSong👉 前端
<script>alert('Hshddh')</script>
1月前   回复

黄裕辉Samuel
最重要还是富文本编辑的时候给转了。这样是有效的防止xxs
1月前   回复

查看更多 >

相关推荐

热 · 专栏 · 人人网FED · 1天前 · Vue.js
一个Vue引发的性能问题

 205  23

热 · 专栏 · 前端小姐姐 · 2天前 · JavaScript
嗨，你真的懂this吗？

 238  80

专栏 · 黄子毅 · 23小时前 · React.js / JavaScript

精读《useEffect 完全指南》

👍 97

💬 9

专栏 · 胡七筒 · 20小时前 · JavaScript / 后端

程序猿生存指南-62 鹊桥银河

👍 36

💬 38

专栏 · 前端小姐姐 · 3天前 · JavaScript

彻底搞懂浏览器Event-loop

👍 29

💬 2

专栏 · zhangxiangliang · 1天前 · JavaScript

每日 30 秒 🕒 数组也会秃顶

👍 12

💬 2

专栏 · Destiny本尊 · 15小时前 · JavaScript

一文搞懂JavaScript原型链（看完绝对懂）

👍 24

💬 5

热 · 专栏 · 前端小姐姐 · 3天前 · JavaScript

9102了，你还不会移动端真机调试？

👍 318

💬 43

热 · 专栏 · OBKoro1 · 4天前 · JavaScript

详解箭头函数和普通函数的区别以及箭头函数的注意事项、不适用场景

👍 235

💬 20

专栏 · toddmark · 1天前 · JavaScript

【译】JavaScript的内存管理和 4 种处理内存泄漏的方法

👍 44

💬 1