web前端安全

1. 浏览器安全策略-同源策略

定义

同源策略是一个重要的安全策略，它用于限制一个origin的文档或者它加载的脚本如何能与另一个源的资源进行交互。它能帮助阻隔恶意文档，减少可能被攻击的媒介。

如果两个 URL 的 protocol、port(如果有指定的话)和host都相同的话，则这两个 URL 是同源。

结构

下表给出了与 URL http://www.abc.com:8080/scripts/jquery.js 的源进行对比的示例：

作用

DOM层面。同源策略限制了来自不同源的JavaScript脚本对当前页面的DOM对象进行读写操作，从而防止跨域脚本篡改DOM结构。

Web数据层面。同源策略限制不同源的站点读取当前站点的Cookie、LocalStorage和IndexDB等数据，从而保障数据的安全性。

网络层面。同源策略限制了通过 XMLHttpRequest 等方式将站点的数据发送给不同源的站点。

跨域还是跨源?

2. 浏览器Cookie策略

Cookie分为两种，一种是Session Cookie，另一种是Third Part Cookie。

Session Cookie是临时会话Cookie，只在浏览器进程的生命周期中有效，它存储在浏览器进程的内存空间中。

Third Part Cookie是本地Cookie，只有Expire过期后才会失效，它存储在本地。默认情况下会跟随符合同源策略的http请求一同发送给后端。

在当前的主流浏览器中，默认会拦截Third=party Cookie的有：IE6、IE7、IE8、safari；不会拦截的有火狐2、火狐3、oprea、Google Chrome、Android等。

W3C添加了P3P header隐私标准，如果网站返回浏览器的Http的头中包含有P3P头，则在某种程度上来说，将允许浏览器发送第三方Cookie。P3P header的介入改变了隐私策略，从而使得iframe，script等标签在IE中不在拦截第三方Cookie的发送。P3P头只需要由网站设置一次即可，之后每次请求都会遵循此策略，而不需要再重复设置。P3P header目前是网站的应用中被广泛应用，因此CSRF的防御中不能依赖于浏览器对第三方Cookie的拦截策略。

3. 跨站脚本攻击(XSS-Cross Site Scripting)

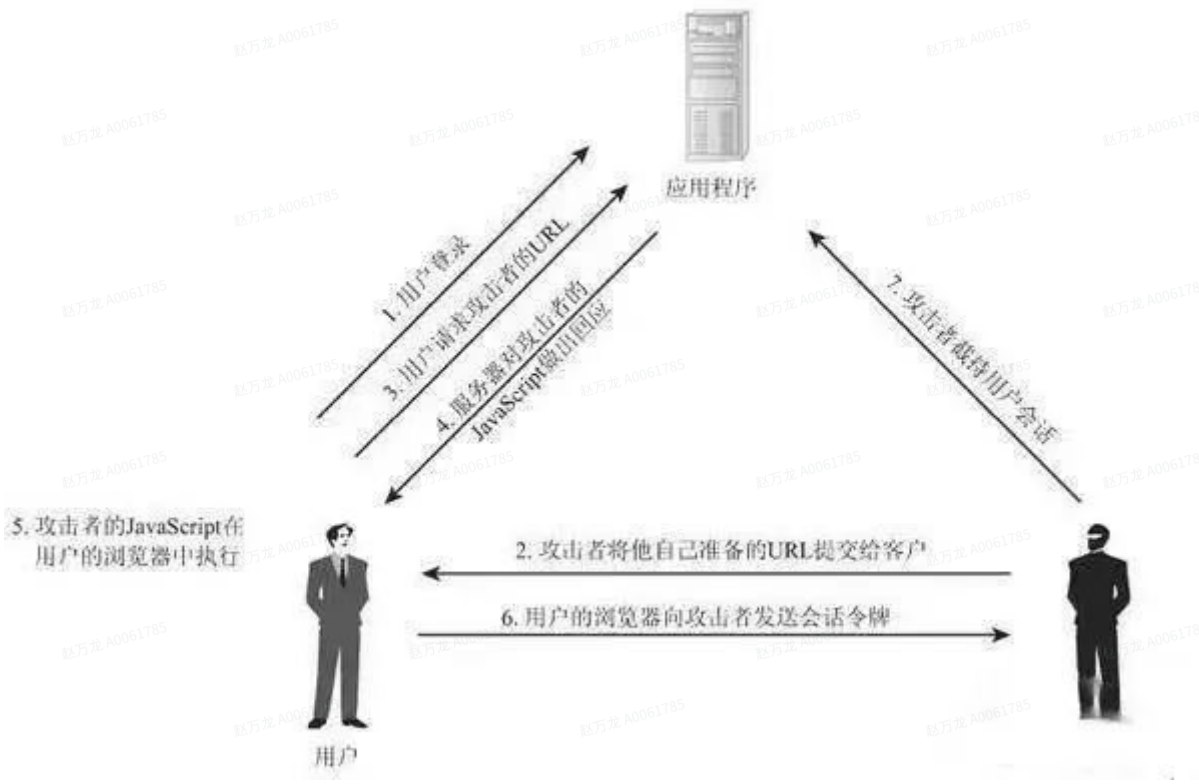
攻击者利用网站漏洞添加恶意代码嵌入到web页面，使得其他用户访问时都会执行嵌入的恶意代码，攻击者就可以突破网站的访问限制并冒充受害者。

反射型XSS

原理

- 攻击者构造出特殊的URL，其中包含恶意代码。
- 用户打开网站时，**服务端将恶意代码从 URL 中取出**，拼接在 HTML 中返回给浏览器。
- 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。



上图中，攻击者（黑衣人）骗取用户信任，构造一个带有跨站脚本的链接，诱骗用户点击（第2、3步），跨站脚本在服务端（WEB应用程序）上没有被过滤，直接返回用户浏览器（第4步），用户浏览器执行恶意脚本（第5步），后面发生的事情就像第6、7步描述的那样。

场景

1. 钓鱼邮件。通常是伪装成授信人员发送一个链接，诱导收信人点击，链接上携带着特殊的执行脚本或恶意代码获取发信人信息，一旦从邮箱中点击了链接就会获取用户信息并传给攻击者指定的服务器。
2. 网站漏洞。对用户输入未做编码就直接渲染成html标签，攻击者会利用这个特性提交带有script脚本的代码，在脚本中执行恶意攻击。

防范

反射型XSS本质上是诱导用户点击特定URL，或是将用户输入原封不动的渲染到html，所以防御措施也很明确。

1. 对html输出字符进行转码，非特定情况不输出html，以text替代。

vue中能用v-text的地方绝不使用v-html

v-html地方必须转码处理

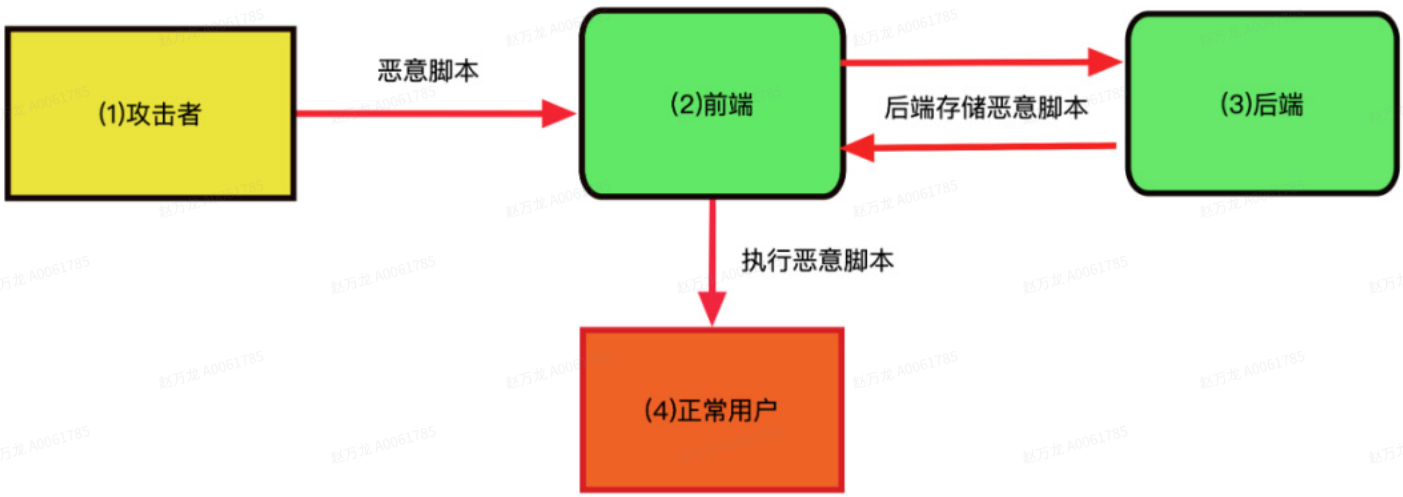
dom中对innerText和innerHtml的使用同上

1. 不点击非信任来源的地址，对特别长的URL尤其要警惕
2. 浏览器包含了基本的xss filter，但应用中也必须处理

存储型XSS

原理

注入型脚本永久存储在目标服务器上。当浏览器请求数据时，脚本从服务器上传回并执行。



1. 攻击者编写恶意攻击的脚本
2. 攻击者访问前端页面，在输入框中输入编写好的恶意脚本
3. 攻击者将恶意脚本进行提交，后端将恶意脚本存储在数据库中
4. 当普通用户访问该网站的时候，该网站会获取存储在数据库中的恶意脚本，但是浏览器不知道它是恶意脚本所以执行了

场景

这种攻击常见于带有用户保存数据的网站功能，如博客、论坛发帖、商品评论、用户私信等。

防范

存储型XSS本质上是通过后端存储攻击者输入的恶意代码，在其他用户页面呈现并执行。

1. 存储前对用户输入转码，不信任任何的用户输入，特别是script标签。
2. 对html输出字符进行转码，非特定情况不输出html，以text替代。
3. cookie设置HttpOnly，配合token或验证码防范。
4. 设置CSP安全策略限制非信任脚本执行。可以通过两种方式设置CSP，一种是meta标签，一种是HTTP响应头Content-Security-Policy

基于DOM的XSS

原理

通过修改原始的客户端代码，受害者浏览器的 DOM 环境改变，导致有效载荷的执行。也就是说，页面本身并没有变化，但由于 DOM 环境被恶意修改，有客户端代码被包含进了页面，并且意外执行。

DOM型XSS和反射型XSS基本相似，不同在于DOM型XSS不经过后端而是前端自身从URL中解析到了恶意代码并嵌入到DOM中执行。

场景

1. 代码中使用了eval('xxx')、innerHTML、document.write来解析用户输入或URL参数
2. 早期版本的jquery，通常是正则过滤表达式存在缺陷，如

使用了$(location.hash)时，在url中添加/#xss <img src='/' onerror='alert("xss")'>

使用html()、append() 或 $('<tag>') 等方法处理用户输入

防范

同反射型XSS

总结

1. 不信任任何用户输入，对用户输入做过滤、转码处理
2. 不使用innerHTML、document.write、eval等方法解析用户输入数据
3. 不直接执行非信任来源的代码，如URL参数
4. 添加必要的安全校验，如token、Authorization

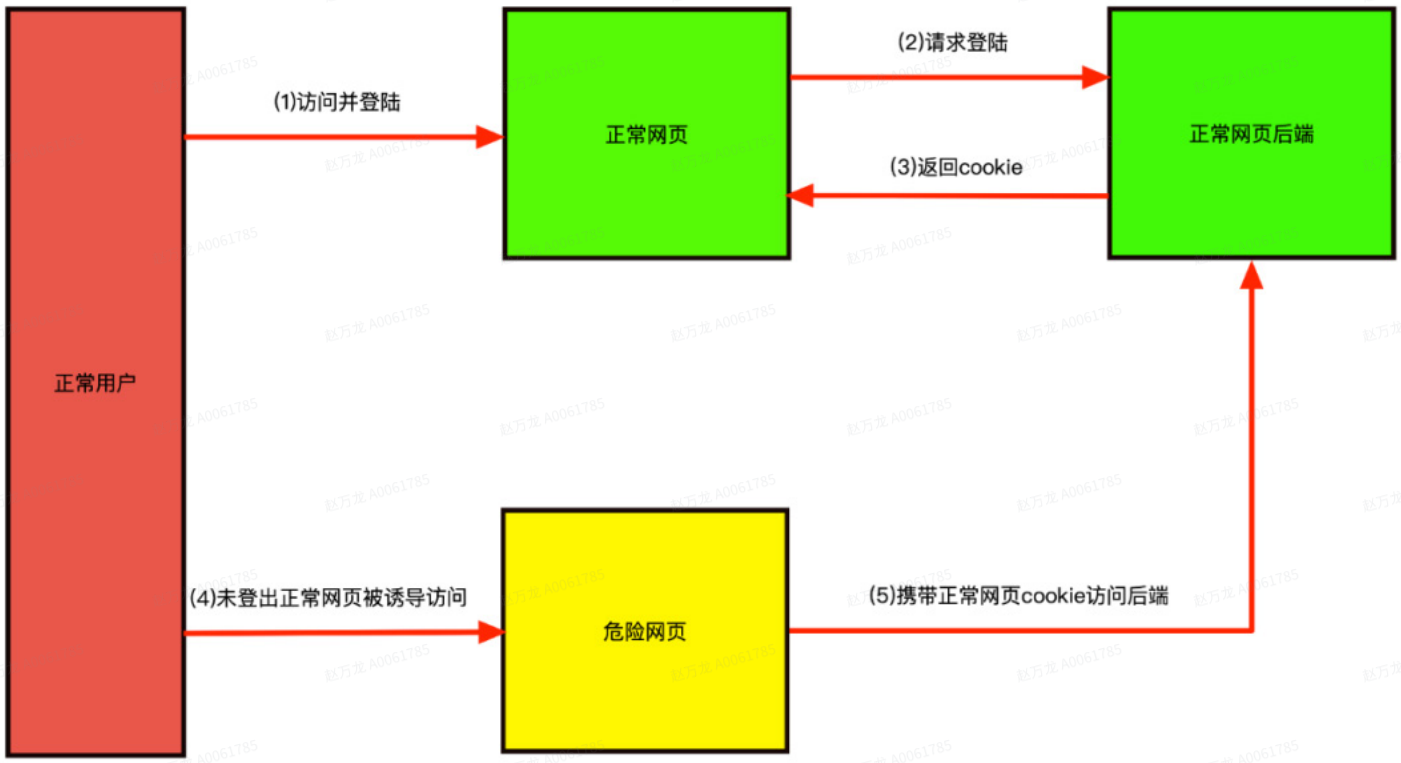
4. 跨站请求伪造攻击(CSRF-Cross Site Request Forgery)

CSRF是一种诱骗受害者提交恶意请求的攻击，攻击者盗用了你的身份，以你的名义发送恶意请求，请求到达后端时，服务器将无法区分恶意请求和合法请求。

CSRF能够做的事情包括：以你名义发送邮件，发消息，盗取你的账号，甚至于购买商品，虚拟货币转账等。

尽管听起来像跨站脚本（XSS），但它与XSS非常不同，并且攻击方式几乎相左。XSS 利用站点内的信任用户，而 CSRF 则通过伪装成来自受信任用户的请求来利用受信任的网站。

根据攻击来源，CSRF分为三种GET类型、POST类型和链接类型。



CSRF产生的两个条件

1. 用户访问了正常网页并产生了cookie

2. 用户没有退出正常网页同时访问了危险网页

Get型CSRF

原理

修改url中的参数直接发起请求

案例

script

link

audio

video

Post型CSRF

原理

自动提交form表单

案例

```
<form method='POST' action='http://user.com/csrf' id="csrfForm">
        <input type='hidden' name='account' value='admin'>
  <input type='hidden' name='amount' value='admin'>
  <input type='submit' value='submit'>
</form>

<script>document.getElementById("csrfForm").submit()</script>
```

链接型CSRF

原理

诱导用户点击链接，通常伪装成用户感兴趣的广告

案例

通常是在论坛发布的图片中嵌入恶意链接，或者以广告的形式诱导用户中招，攻击者通常会以比较夸张的词语诱骗用户点击，例如：

<a href="" taget="_blank"> <a/>

由于之前用户登录了信任的网站A，并且保存登录状态，只要用户主动访问这个页面，则表示攻击成功。

防范策略

在业界目前防御 CSRF 攻击主要有四种策略：

1. 验证 HTTP Referer 字段；
2. 请求地址中添加 token 并验证；
3. HTTP 头中自定义属性并验证(签名)；
4. Chrome 浏览器端启用 SameSite cookie

验证 HTTP 字段

根据 HTTP 协议，在 HTTP 头中有一个字段叫 Referer，它记录了该 HTTP 请求的来源地址。比如从 https://image.baidu.com 中显示一张https://user.com/image.png 时，图片的请求上会携带 Referer=https://image.baidu.com。因此，要防御 CSRF 攻击，后端只需要对于每一个转账请求验证其 Referer 值，如果是以user.com 开头的域名，则说明该请求是来自银行网站自己的请求，是合法的。如果 Referer 是其他网站的话，则有可能是黑客的 CSRF 攻击，拒绝该请求。

这种方法的显而易见的好处就是简单易行，网站的普通开发人员不需要操心 CSRF 的漏洞，只需要在最后给所有安全敏感的请求统一增加一个拦截器来检查 Referer 的值就可以。特别是对于当前现有的系统，不需要改变当前系统的任何已有代码和逻辑，没有风险，非常便捷。

然而，这种方法并非万无一失。Referer 的值是由浏览器提供的，虽然 HTTP 协议上有明确的要求，但是每个浏览器对于 Referer 的具体实现可能有差别，并不能保证浏览器自身没有安全漏洞。使用验证 Referer 值的方法，就是把安全性都依赖于第三方（即浏览器）来保障，从理论上来讲，这样并不安全。事实上，对于某些浏览器，比如 IE6 或 FF2，目前已经有一些方法可以篡改 Referer 值。如果 bank.example 网站支持 IE6 浏览器，黑客完全可以把用户浏览器的 Referer 值设为以 bank.example 域名开头的地址，这样就可以通过验证，从而进行 CSRF 攻击。

即便是使用最新的浏览器，黑客无法篡改 Referer 值，这种方法仍然有问题。因为 Referer 值会记录下用户的访问来源，有些用户认为这样会侵犯到他们自己的隐私权，特别是有些组织担心 Referer 值会把组织内网中的某些信息泄露到外网中。因此，用户自己可以设置浏览器使其在发送请求时不再提供 Referer。当他们正常访问银行网站时，网站会因为请求没有 Referer 值而认为是 CSRF 攻击，拒绝合法用户的访问。

请求地址中添加 token 并验证

CSRF 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于 cookie 中，因此黑客可以在不知道这些验证信息的情况下直接利用用户的 cookie 来通过安全验证。要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 cookie 之中。可以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有 token 或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。

这种方法要比检查 Referer 要安全一些，token 可以在用户登陆后产生并放于 session 之中，然后在每次请求时把 token 从 session 中拿出，与请求中的 token 进行比对，但这种方法的难点在于如何把 token 以参数的形式加入请求。对于 GET 请求，token 将附在请求地址之后，这样 URL 就变成

```
1 http://url?csrftoken=tokenvalue
```

而对于 POST 请求来说，要在 form 的最后加上

```
1  <input type="hidden" name="csrftoken" value="tokenvalue"/>
```

该方法有一个缺点是难以保证 token 本身的安全。特别是在一些论坛之类支持用户自己发表内容的网站，黑客可以在上面发布自己个人网站的地址。由于系统也会在这个地址后面加上 token，黑客可以在自己的网站上得到这个 token，并马上就可以发动 CSRF 攻击。为了避免这一点，系统可以在添加 token 的时候增加一个判断，如果这个链接是链到自己本站的，就在后面添加 token，如果是通向外网则不加。不过，即使这个 csrftoken 不以参数的形式附加在请求之中，黑客的网站也同样可以通过 Referer 来得到这个 token 值以发动 CSRF 攻击。这也是一些用户喜欢手动关闭浏览器 Referer 功能的原因。

HTTP 头中自定义属性并验证(签名)

这种方法也是使用 token 并进行验证，和上一种方法不同的是，这里并不是把 token 以参数的形式置于 HTTP 请求之中，而是把它放到 HTTP 头中自定义的属性里。通过 XMLHttpRequest 这个类，可以一次性给所有该类请求加上 csrftoken 这个 HTTP 头属性，并把 token 值放入其中。这样解决了上种方法在请求中加入 token 的不便，同时，通过 XMLHttpRequest 请求的地址不会被记录到浏览器的地址栏，也不用担心 token 会透过 Referer 泄露到其他网站中去。

然而这种方法的局限性非常大。XMLHttpRequest 请求通常用于 Ajax 方法中对于页面局部的异步刷新，并非所有的请求都适合用这个类来发起，而且通过该类请求得到的页面不能被浏览器所记录下，从而进行前进，后退，刷新，收藏等操作，给用户带来不便。另外，对于没有进行 CSRF 防护的遗留系统来说，要采用这种方法来进行防护，要把所有请求都改为 XMLHttpRequest 请求，这样几乎是要重写整个网站，这代价无疑是不能接受的。

Chrome 浏览器端启用 SameSite cookie

原本的 Cookie 的 header 设置是长这样：

```
1 Set-Cookie: session_id=esadfas325
```

需要在尾部增加 SameSite 就好：

```
1 Set-Cookie: session_id=esdfas32e5; Secure; SameSite=None
```

SameSite 有三种模式，None、Lax跟Strict模式，默认启用Strict模式，可以自己指定模式，当指定None时必须添加Secure：

```
1 Set-Cookie: session_id=esdfas32e5; SameSite=Strict
2 Set-Cookie: foo=bar; SameSite=Lax
```

Strict模式规定 cookie 只允许相同的site使用，不应该在任何的 cross site request 被加上去。即a标签、form表单和XMLHttpRequest提交的内容，只要是提交到不同的site去，就不会带上cookie。

但也存在不便，例如朋友发送过来我已经登陆过的一个页面链接，我点开后，该页面仍然需要重新登录。

有两种处理办法，第一种是与Amazon一样，准备两组不同的cookie，第一组用于维持登录状态不设定SameSite，第二组针对的是一些敏感操作会用到（例如购买、支付、设定账户等）严格设定SameSite。

基于这个思路，就产生了 SameSite 的另一一种模式：Lax模式。

Lax 模式打开了一些限制，例如

```
1 <a>
```

```
2 <link rel="prerender">
3 <form method="GET">
```

这些都会带上cookie。但是 POST 方法 的 form，或是只要是 POST, PUT, DELETE 这些方法，就不会带cookie。

但一定注意将重要的请求方式改成POST，否则GET仍然会被攻击。

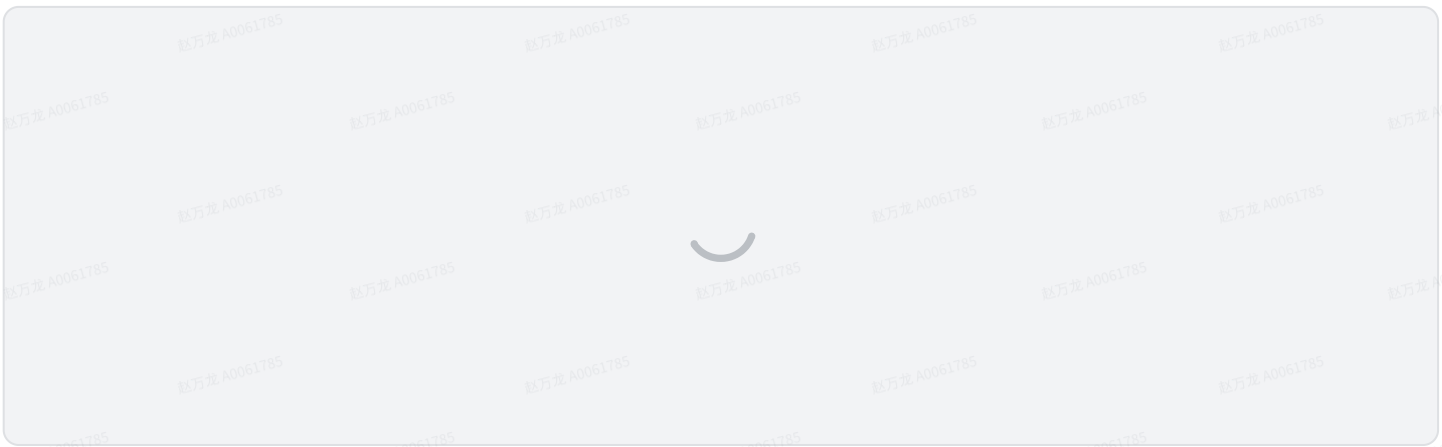
总结:

SameSite 可以有下面三种值：

1. **Strict** 仅允许一方请求携带 Cookie，即浏览器将只发送相同站点请求的 Cookie，即当前网页 URL 与请求目标 URL 完全一致。
2. **Lax** 允许部分第三方请求携带 Cookie。
3. **None** 无论是否跨站都会发送 Cookie。

之前默认是 None 的，Chrome80 后默认是 Lax。

Lax的情况见下表：



HTTP 接口不支持 SameSite=none

部分浏览器不支持部分SameSite=none

5. URL跳转漏洞

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。

比如知乎中的文章链接：

https://zhuanlan.zhihu.com/p/405710228

JS安全之路：用JS对JS代码混淆加密

JS代码安全之路：用JS对JS代码混淆加密作者： http://JShaman.com：w2sft内容预告：本文将实例讲解以下JS代码混淆加密技术： 方法名转义和转码、成员表达式转IIFE、函数标准化、数值混淆、布尔型常量值混淆、二进…

6. 点击劫持(Click Jacking)

ClickJacking点击劫持，也叫UI覆盖攻击，攻击者会利用一个或多个透明或不透明的层来诱骗用户支持点击按钮的操作，而实际的点击确实用户看不到的一个按钮，从而达到在用户不知情的情况下实施攻击。

iframe覆盖

这种攻击方式的关键在于可以实现页中页的<iframe>标签，并且可以使用css样式表将他不可见。

防止点击劫持有两种主要方法：X-FRAME-OPTIONS

X-FRAME-OPTIONS是微软提出的一个http响应首部，指示浏览器不允许从其他域进行取景，专门用来防御利用iframe嵌套的点击劫持攻击。并且在IE8、Firefox3.6、Chrome4以上的版本均能很好的支持。

DENY : 拒绝任何域加载

SAMEORIGIN : 允许同源域下加载

ALLOW-FROM : 可以定义允许frame加载的页面地址

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options

X-Frame-Options - HTTP | MDN

The X-Frame-Options HTTP response header can be used to indicate whether or not a browser should be allowed to render a page in a <frame>, <iframe>,…

图片覆盖

图片覆盖攻击（Cross Site Image Overlaying），攻击者使用一张或多张图片，利用图片的style或者能够控制的CSS，将图片覆盖在网页上，形成点击劫持。当然图片本身所带的信息可能就带有欺骗的含义，这样不需要用户点击，也能达到欺骗的目的。

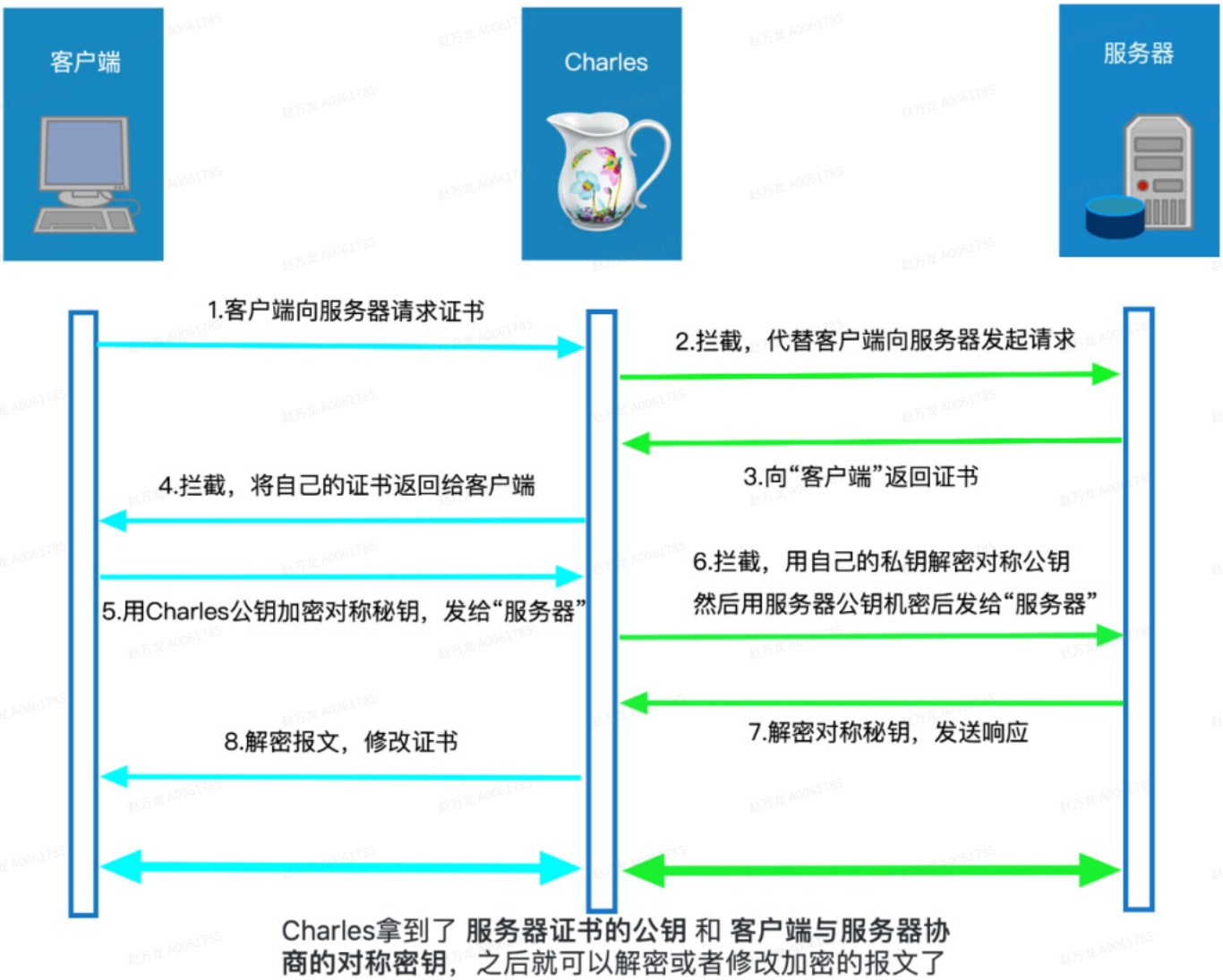
7. SQL注入

不展开...

8. 网络传输安全

中间人 (Man-in-the-middle attack, MITM) 是指攻击者与通讯的两端分别创建独立的联系, 并交换其所收到的数据, 使通讯的两端认为他们正在通过一个私密的连接与对方直接对话, 但事实上整个会话都被攻击者完全控制. 在中间人攻击中, 攻击者可以拦截通讯双方的通话并插入新的内容。

是不是觉得有了https网络传输安全问题就迎刃而解了呢，即使被中间人拦截了，数据也是加密的。其实不是这样的，不知道大家有没有使用过charles进行抓包呢，如果数据都是加密的，为啥charles抓包后我们能够看到传输的明文呢，其实这就是中间人攻击。



防范

1. 接口加密
2. 接口验签
3. 接口防重复提交
4. 检测客户端环境
5. 验证码
6. 客户端代码混淆加密，增加解析签名难度

HTTPS就一定安全吗?