2021大厂前端秋季招聘面试（社招+校招）核心面试点(三)

#2021#

前端安全

你了解哪些前端安全相关的知识? 或者名词?

浏览器相关:

1. XSS
2. CSRF
3. HTTPS
4. CSP (内容安全策略, 可以禁止加载外域代码, 禁止外域提交等等)
5. HSTS (强制客户端使用HTTPS与服务端建立连接)
6. X-Frame-Options (控制当前页面是否可以被嵌入到Ifrmae中)
7. SRI (subresource intergrity 子资源完整性, 前端可以用webpack-subresource-integrity插件, 在每个script上添加hash值, 校验加载的资源是否和当时打包生成的一致)
8. Referrer-Policy (控制referer的携带策略)

Node(服务端)相关

1. 本地文件操作相关：路径拼接导致的文件泄露
2. ReDOS
3. 时序攻击
4. ip origin referrer等request headers的校验(在做爬虫应用的时候对此会有深刻的体会)

能稍微详细的聊一下XSS吗?

Cross-site scripting, 跨站脚本, 通常简称为XSS.

> 为什么简写为XSS而不是CSS？因为CSS被广泛应用于样式的称呼里，而Cross意味交叉，X字母正好是符合交叉的含义，所以简称为XSS。

说白了就是攻击者想尽一切办法将可执行代码注入到网页中, 而恶意代码未经过滤，与网站正常的代码混在一起；浏览器无法分辨哪些脚本是可信的，导致恶意脚本被执行。

外在表现上, 都有哪些攻击场景?

1. 评论区植入js代码(即可输入的地方植入js代码)
2. url上拼接js代码

- TIPS: 有点同学可能觉得在这种场景下, 用户能输入的代码长度有限, 根本构不成什么威胁? 然而攻击者是可以通过引入外部脚本来实现复杂攻击的.

具体从技术角度上分析, 都有哪些xss攻击的类型呢?

1. 存储型 Server

- 场景：常见于带有用户保存数据的网站功能，攻击者通过可输入区域来注入恶意代码, 如论坛发帖、商品评论、用户私信等。
- 攻击步骤：
  1. 攻击者将恶意代码提交到目标网站的数据库中
  2. 用户打开目标网站时，服务端将恶意代码从数据库中取出来，拼接在HTML中返回给浏览器(因为用户之间是可以相互看到帖子、评论等的)
  3. 用户浏览器在收到响应后解析执行，混在其中的恶意代码也同时被执行
  4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户行为，调用目标网站的接口执行攻击者指定的操作。

2. 反射型 Server

与存储型的区别在于，存储型的恶意代码通过可输入区域, 存储在数据库中，而反射型的恶意代码拼接在URL上。
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。

- 场景：通过 URL 传递参数的功能，如网站搜索、跳转等。
- 攻击步骤：
  1. 攻击者构造出特殊的 URL，其中包含恶意代码。
  2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。
  3. 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。
  4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站的接口执行攻击者指定的操作。

3. Dom型 Browser

DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全

漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

- 场景：通过 URL 传递参数的功能，如网站搜索、跳转等。
- 攻击步骤：
  1. 攻击者构造出特殊的 URL，其中包含恶意代码。
  2. 用户打开带有恶意代码的 URL。
  3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行。
  4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站的接口执行攻击者指定的操作。

简单模拟一下Dom型XSS攻击?

1. index.html

```
<!DOCTYPE html>
<html lang="zh">
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=edge" />
        <meta
            name="viewport"
            content="width=device-width, initial-scale=1, minimum-scale=1,
maximum-scale=1, user-scalable=no"
        />
        <title>XSS</title>
    </head>
    <body>
        <a href="">跳转到新地址</a>
    </body>
    <script src="type-dom.js"></script>
</html>
```

2. type-dom.js

```
const a = document.getElementsByTagName('a')[0];
const queryObject = {};
const search = window.location.search;
```

```
search.replace(/([^&=?]+)=([^&]+)/g, (m, $1, $2) => (queryObject[$1] =
decodeURIComponent($2)));

a.href = queryObject.redirectUrl;
```

3. 打开当前index.html, 添加参数redirectUrl访问

   比如

```
redirectUrl=javascript:alert('哈哈哈笨蛋!!')
```

4. 点击a链接发现已经被xss攻击了.

5. 有的同学说这样太简单了, 你难不成可以输入非常长的代码在链接上? 长度有限制的吧?

   同学说的对, 但是我们上面讲过, 攻击者可以直接加载一个js文件.

   比如我们写这样一段脚本, 目的是创建一个script标签并且引入remote.js文件, 将这一串代码作为redirectUrl的值访问链接.

```
var script = document.createElement('script');
script.type = 'text/javascript';
script.async = true;
script.src = 'remote.js';
var s = document.getElementsByTagName('script')[0];
s.parentNode.insertBefore(script, s);
```

   新建remote.js文件, 来尝试耗性能的代码. 咱们因为在直播, 就设置一个count来停止吧, 否则浏览器可能会崩掉.

```
let count = 0;
console.log(window.navigator.userAgent);

while(count++ < 100) {
    console.log('我要通过巨量运算把你搞崩溃')
}
```

6. 而如果我们网站的cookie里有重要的用户信息, 那么攻击者是否就可以通过document.cookie获取到了?\

   比如我们在原来的网站脚本type-dom.js里设置一下cookie, 然后在攻击脚本remote.js里获取.

   ```
   document.cookie="name=lubai12313";
   ```

   ```
   console.log(document.cookie);
   ```

   但是聪明的同学应该也想到了, 只要我们给重要的cookie设置了http-only, 就算被dom xss攻击了, 攻击者也造不成大的影响.

7. 再来试试访问url就直接出发的xss攻击.

   type-dom.js

   ```
   document.write(queryObject.name)
   ```

   添加参数访问url,

   ```
   name=<script>window.alert(1)</script>
   ```

8. 可以试一下这个网站, https://alf.nu/alert1

   如何防范XSS攻击呢?

   主旨：防止攻击者提交恶意代码，防止浏览器执行恶意代码

1. 对数据进行严格的输出编码：如HTML元素的编码，JS编码，CSS编码，URL编码等等
   - 避免拼接 HTML；Vue/React 技术栈，避免使用 v-html / dangerouslySetInnerHTML
2. CSP HTTP Header，即 Content-Security-Policy（不支持CSP的旧版浏览器可以设置X-XSS-Protection）
   - 增加攻击难度，配置CSP(本质是建立白名单，由浏览器进行拦截)
   - Content-Security-Policy: default-src 'self' -所有内容均来自站点的同一个源（不包括其子域名）
   - Content-Security-Policy: default-src 'self' *.trusted.com -允许内容来自信任的域名及其子域名 (域名不必须与CSP设置所在的域名相同)
   - Content-Security-Policy: default-src https://lubai.com -该服务器仅允许通过HTTPS方式

并仅从lubai.com域名来访问文档

> 可以做到很多事情，比如：
>
> 禁止加载外域代码，防止复杂的攻击逻辑。
>
> 禁止外域提交，网站被攻击后，用户的数据不会泄露到外域。

3. 输入验证：比如一些常见的数字、URL、电话号码、邮箱地址等等做校验判断
4. 开启浏览器XSS防御：Http Only cookie，禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。
5. 验证码

那再来说一下CSRF吧?

Cross-site request forgery, 跨站请求伪造.

攻击者诱导受害者进入恶意网站，在第三方网站中，向被攻击网站发送跨站请求。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，达到冒充用户对被攻击的网站执行某项操作的目的。

攻击步骤

1. 受害者登录 a.com，并保留了登录凭证（Cookie）
2. 攻击者引诱受害者访问了b.com
3. b.com 向 a.com 发送了一个请求：a.com/act=xx浏览器会默认携带a.com的Cookie
4. a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求
5. a.com以受害者的名义执行了act=xx
6. 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作

攻击类型

- GET型：如在页面的某个 img 中发起一个 get 请求

```
<img src="http://bank.example/withdraw?name=xxx&amount=xxxx" />
```

- POST型：通过自动提交表单到恶意网站

```
<form action="http://bank.example/withdraw" method=POST>
    <input type="hidden" name="account" value="lubai" />
    <input type="hidden" name="amount" value="10000" />
    <input type="hidden" name="for" value="hacker" />
</form>
<script> document.forms[0].submit(); </script>
```

```
<a href="http://bank.example/withdraw?name=xxx&amount=xxxx" taget="_blank">
错过再等一点！！！快来看看
</a>
```

如何防范CSRF的攻击呢?

首先咱们通过上面的举例可以知道, CSRF一般都是发生在第三方域名, 攻击者也无法获取到Cookie信息, 只是可以利用浏览器机制去使用Cookie.

所以咱们可以针对这两点来看防范策略:

阻止第三方域名的访问

1. Cookie SameSite

   SameSite有3个值： Strict, Lax和None

- Strict：浏览器会完全禁止第三方cookie。比如a.com的页面中访问 b.com 的资源，那么a.com中的cookie不会被发送到 b.com服务器，只有从b.com的站点去请求b.com的资源，才会带上这些Cookie
- Lax：在跨站点的情况下，从第三方站点链接打开和从第三方站点提交 Get方式的表单这两种方式都会携带Cookie。但如果在第三方站点中使用POST方法或者通过 img、Iframe等标签加载的URL，这些场景都不会携带Cookie
- None：任何情况下都会发送 Cookie数据

2. 同源检测

   通过检测request header中的origin referer等, 来确定发送请求的站点是否是自己期望中站点.

比如referer, 我们可以在服务端去判断referer是否来自可信域, 同样也可以做一些referer发送时的设置：

对于同源的链接和引用，会发送Referer，referer值为Host不带Path；跨域访问则不携带Referer。例如：aaa.com引用bbb.com的资源，不会发送Referer(服务端在接收到没有referer的请求时就不做响应)

这就提到了Referrer-Policy这个请求头. https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Referrer-Policy, 控制什么情况下应该携带/不携带referer

提交请求时附加额外信息

因为攻击者无法通过csrf来获取本域下的cookie等信息, 所以可以利用这一点来做防范.

1. CSRF Token

- 用户打开页面的时候, 服务器利用加密算法给当前用户生成一个ToKen
- 每次页面加载时, 前端把获取到的Token加到所有的能发请求的html元素上, 比如form, a
- 每次前端发起请求, 都携带Token参数
- 服务端每次接收请求, 都校验Token的有效性.

2. 双重Cookie

- 用户访问网站的时候, 服务器向浏览器注入一个额外的cookie, 内容随便, 比如csrfcookie=lubaixzxfasdfasfew
- 每次前端发起请求, 都在请求上拼接上csrfcookie这个参数, 参数值就从cookie里获取
- 服务端每次收到请求, 就去校验请求参数里的值和cookie里的值是否一致

但是这种方式安全性不如CSRF Token. 咱们来看一下原因：

1. 如果前端域名和服务端域名不一样, 比如前端 fe.a.com, 后端 rd.a.com, 那如果服务端希望前端能拿到csrfcookie, 就只能把这个cookie设置到a.com域下, 并且不能设置为http-only
2. 那么a.com的每个子域名就都可以获取到这个cookie
3. 一旦某个子域名遭到xss攻击, cookie就很容易被窃取或者被篡改.
4. 攻击者利用篡改或者窃取的csrfcookie, 就可以攻击fe.a.com了

Node(服务端)相关的安全问题有了解过吗? 大概举几个例子?

本地文件操作

比如我们提供一个静态服务, 通过请求的参数url来返回给用户或者前端想要的资源.

1. 新建文件node/static-sever-dangerous.js

```
const fs = require('fs');
const http = require('http');
const path = require('path');

http
  .createServer(function (req, res) {
    const file = path.join(__dirname, 'static', req.url);

    fs.readFile(file, function (err, data) {
      if (err) {
        res.writeHead(404, { "Content-Type": "text/plain;charset=utf-8" });
        res.end('找不到对应的资源');
        return;
      }
      res.writeHead(200, { "Content-Type": "text/plain;charset=utf-8" });
      res.end(data);
    });
  })
  .listen(8080);
  console.log('server listening on port 8080');
```

2. 新建文件夹static, 里面随便放点文件, 提供给外界请求

   比如 test.json

```
{
    "name": "lubai"
}
```

3. 启动服务, 本地请求url

   http://localhost:8080/test.json

http://localhost:8080/test1.json

没有任何问题对吧? 存在的资源正常返回了, 不存在的资源返回404了.

4. 请求一个不一样的url

- http://localhost:8080/?/../../../questions.md
- http://localhost:8080/?/../../../private.js

同级新建一个private.js, 里面写上一些私密信息

```
// 这是一个私密文件
```

可以看到, 攻击者可以通过拼接相对路径, 一次次猜你项目的结构, 并且可以访问到你服务器上的各种资源!

5. 怎么解决这个问题?

很多node框架都自带插件来屏蔽这个问题的发生

比如express.static, koa-static, 当然也有第三方包支持(resolve-path)

咱们用resove-path来解决一下这个问题.

- npm init
- yarn add resolve-path –registry=https://registry.npm.taobao.org
- 新建 static-sever-secure.js 文件

```
const fs = require('fs');
const http = require('http');
const path = require('path');
// 引入resolve-path
const resolvePath = require('resolve-path');

http
```

```
  .createServer(function (req, res) {
    try {
        // 先获取rootDir
        const rootDir = path.join(__dirname, 'static');
        // 调用resolvePath
        const file = resolvePath(rootDir, req.url);

        fs.readFile(file, function (err, data) {
          if (err) {
            // 把错误抛出去
            throw err;
          }
          res.writeHead(200, { "Content-Type": "text/plain;charset=utf-8" });
          res.end(data);
        });
    } catch(e) {
        // catch住错误，防止服务直接挂掉
        console.log(e);
        res.writeHead(404, { "Content-Type": "text/plain;charset=utf-8" });
        res.end('找不到对应的资源');
    }

  })
  .listen(8081); // 改成8081端口
  console.log('server listening on port 8081');
```

1. 可以看到resolve-path的源码对path做了严格的限制.

截图服务

比如咱们来实现一个简单的截图服务

1. 安装好必要的npm包

```
yarn add puppeteer-chromium-resolver koa --registry=https://registry.npm.taobao.org
```

2. 新建 screen-shot.js

```
(async () => {
```

```
  const PCR = require('puppeteer-chromium-resolver');
  const stats = await PCR();

  const browser = await stats.puppeteer
    .launch({
      headless: true,
      args: ['--no-sandbox'],
      executablePath: stats.executablePath,
    })
    .catch(function (error) {
      console.log(error);
    });

  const Koa = require('koa');
  const app = new Koa();

  app.use(async ctx => {
    const { url } = ctx.query;

    // 这里演示不合理的Url校验
    if (!url) {
      ctx.body = 'Invalid url';

      return;
    }

    ctx.set('Content-Type', 'image/png');

    const page = await browser.newPage();
    await page.goto(url, { waitUntil: 'networkidle2' });

    ctx.body = await page.screenshot({ encoding: 'binary', type: 'png' });

    await page.close();
  });
  app.listen(8083);

  console.log('server listening on port 8083');
})();
```

3. 访问url

- http://localhost:8083/?url=http://localhost:8080/test.json

  这里咱们传入url为之前启动的不安全的静态服务. 可以看到我们直接截图了对应的资源

- http://localhost:8083/x?url=file:///etc/zshrc

  这里咱们传入url为文件系统的zshrc, 可以发现我们直接截图了系统文件的内容.

  ReDOS

  新建redos.js, 分别看一下这三个例子的执行时间

```
console.time('case-1');
// 能够匹配成功
/A(B|C+)+D/.test('ACCCCCCCCCCCCCCCCCCCCCCCCCCCD');
console.timeEnd('case-1');

console.time('case-2');
// 不能匹配成功
/A(B|C+)+D/.test('ACCCCCCCCCCCCCCCCCCCCCCCCCCCX');
console.timeEnd('case-2');

console.time('case-3');
// 不能匹配成功
/A(B|C+)+D/.test(`A${'C'.repeat(30)}X`);
console.timeEnd('case-3');
```

  可以看到, 当不能匹配成功的时候, 每多一个字符, 所消耗是时间都是指数增长的.

  因为咱们服务器经常会有正则去匹配一些传入的参数, 所以攻击者就可以利用正在表达式的这个特性, 来一直占用服务器运算资源, 造成服务器宕机.

  具体原理可以看这篇文章：https://snyk.io/node-js/connect

正则表达式一般情况下会去匹配第一种可能性, 比如一个正确的字符串 ACCCD, 那么直接匹配到最后发现成功了, 耗时就很短.

而比如ACCCX这样一个字符, 每当一次匹配不成功, 就会尝试回溯到上一个字符, 看看能不能有其他的组合来匹配到这个字符串.

比如刚才说的ACCCX这样一个字符串, 会去尝试匹配四种不同的"C"字母组合来与其他字母组合, 看是否符合条件.

1. CCC
2. CC+C
3. C+CC
4. C+C+C.

可以在写完正则后去这个网址测试一下 https://regex.rip/? 测试是否会遭到reDos.

时序攻击

这种攻击方式在咱们编码过程中可能很少见, 看下面这个例子

比如咱们要匹配接收的数组和咱们定义好的数组是否完全一致, 如果一致才可以进行之后的操作.

```
function compareArray(realArray, userInputArrary) {
    for (let i = 0; i < realArray.length; i++) {
        if (realArray[i] !== userInputArray[i]) {
            return false;
        }
    }
    return true;
}
```

这样写有没有什么问题? 逻辑上没有任何问题, 但是有安全问题.

因为我们如果判断到一个字符不相等, 就提前返回了！！

这给攻击者提供了一种方式, 就是根据服务器的响应时间来碰撞出realArray的值.

比如realArray = [2,3,6,1]

攻击者开始尝试
inputArray = [1,2]
inputArray = [1,3]

发现这两种的响应时间几乎一致, 则可以认为第一个数字不是1

inputArray = [2, 2]

发现响应时间延长了, 则可以认为第一个数字是2.

长而久之, 就可以碰撞出真实的realArray了.

当然这里只是举个简单的例子, 真实的时序攻击和程序肯定都不是这么简单, 这里只是跟大家科普一下这种安全问题.

事件循环

1. 事件循环存在的意义是什么?
2. 事件循环的基本概念?
3. 浏览器和node环境的事件循环有什么区别?
4. 来看一下这几个代码片段的输出结果是什么?

为什么有事件循环?

单线程:
JavaScript的主要用途是与用户互动，以及操作DOM。如果它是多线程的会有很多复杂的问题要处理，比如有两个线程同时操作DOM，一个线程删除了当前的DOM节点，一个线程是要操作当前的DOM阶段，最后以哪个线程的操作为准？为了避免这种，所以JS是单线程的。即使H5提出了web worker标准，它有很多限制，受主线程控制，是主线程的子线程。

非阻塞：通过 event loop 实现。

宏任务和微任务

宏任务和微任务
为什么要引入微任务，只有一种类型的任务不行么?

页面渲染事件，各种IO的完成事件等随时被添加到任务队列中，一直会保持先进先出的原则执

行，我们不能准确地控制这些事件被添加到任务队列中的位置。但是这个时候突然有高优先级的任务需要尽快执行，那么一种类型的任务就不合适了，所以引入了微任务队列。

浏览器里的事件循环

关于微任务和宏任务在浏览器的执行顺序是这样的：

1. 执行全局Script同步代码，这些同步代码有一些是同步语句，有一些是异步语句（比如setTimeout等）；
2. 全局Script代码执行完毕后，调用栈Stack会清空；
3. 从微队列microtask queue中取出位于队首的回调任务，放入调用栈Stack中执行，执行完后microtask queue长度减1；
4. 继续取出位于队首的任务，放入调用栈Stack中执行，以此类推，直到直到把microtask queue中的所有任务都执行完毕。注意，如果在执行microtask的过程中，又产生了microtask，那么会加入到队列的末尾，也会在这个周期被调用执行；
5. microtask queue中的所有任务都执行完毕，此时microtask queue为空队列，调用栈Stack也为空；
6. 取出宏队列macrotask queue中位于队首的任务，放入Stack中执行；
7. 执行完毕后，调用栈Stack为空；
8. 重复第3-7个步骤；
9. 重复第3-7个步骤；

……

可以看一下这个网站, 理解一些调用栈和队列的概念. http://latentflip.com/loupe

常见的 task（宏任务） 比如：setTimeout、setInterval、script（整体代码）、I/O 操作、UI 渲染等。

常见的 micro-task 比如: new Promise().then(回调)、MutationObserver(html5新特性) 等。

宏任务队列里一次循环是要执行所有任务, 还是只执行一个?

宏队列macrotask一次只从队列中取一个任务执行，执行完后就去执行微任务队列中的任务；

微任务队列里一次循环是要执行所有任务, 还是只执行一个?

微任务队列中所有的任务都会被依次取出来执行，知道microtask queue为空；

代码输出顺序题

1.js

2.js

3.js

Node里的事件循环

大体的task（宏任务）执行顺序是这样的：

- timers定时器：本阶段执行已经安排的 setTimeout() 和 setInterval() 的回调函数。
- pending callbacks待定回调：执行延迟到下一个循环迭代的 I/O 回调。
- idle, prepare：仅系统内部使用。
- poll 轮询：检索新的 I/O 事件;执行与 I/O 相关的回调（几乎所有情况下，除了关闭的回调函数，它们由计时器和 setImmediate() 排定的之外），其余情况 node 将在此处阻塞。
- check 检测：setImmediate() 回调函数在这里执行。
- close callbacks 关闭的回调函数：一些准备关闭的回调函数，如：socket.on('close', ...)。

微任务和宏任务在Node的执行顺序

1. Node 10以前：
   执行完一个阶段的所有任务
   执行完nextTick队列里面的内容
   然后执行完微任务队列的内容

2. Node 11以后：
   和浏览器的行为统一了，都是每执行一个宏任务就执行完微任务队列。

算法 - 接雨水

敬请期待~课上有时间就讲