网络

一. http协议

HTTP 的特性

- HTTP 协议构建于 TCP/IP 协议之上，是一个应用层协议，默认端口号是 80
- HTTP 是无连接无状态的

HTTP 报文

请求报文

HTTP 协议是以 ASCII 码传输，建立在 TCP/IP 协议之上的应用层规范。规范把 HTTP 请求分为三个部分：状态行、请求头、消息主体。类似于下面这样：

```
x

<method> <request-URL> <version>

<headers>


<entity-body>
```

HTTP 定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE。URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资源，而 HTTP 中的GET，POST，PUT，DELETE就对应着对这个资源的查，增，改，删4个操作。

1. GET 用于信息获取，而且应该是安全的 和 幂等的。

   所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET 请求一般不应产生副作用。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。

   幂等的意味着对同一 URL 的多个请求应该返回同样的结果。

   GET 请求报文示例：

   ```
   xxxxxxxxxx
    GET /books/?sex=man&name=Professional HTTP/1.1
    Host: www.example.com
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
    Gecko/20050225 Firefox/1.0.1
    Connection: Keep-Alive
   ```

2. POST 表示可能修改变服务器上的资源的请求。

   ```
   xxxxxxxxxx
    POST / HTTP/1.1
   ```

```
Host: www.example.com

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)

Gecko/20050225 Firefox/1.0.1

Content-Type: application/x-www-form-urlencoded

Content-Length: 40

Connection: Keep-Alive


sex=man&name=Professional
```

3. 注意:

    - GET 可提交的数据量受到URL长度的限制，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制
    - 理论上讲，POST 是没有大小限制的，HTTP 协议规范也没有进行大小限制，出于安全考虑，服务器软件在实现时会做一定限制
    - 参考上面的报文示例，可以发现 GET 和 POST 数据内容是一模一样的，只是位置不同，一个在 URL 里，一个在 HTTP 包的包体里

POST 提交数据的方式

HTTP 协议中规定 POST 提交的数据必须在 body 部分中，但是协议中没有规定数据使用哪种编码方式或者数据格式。实际上，开发者完全可以自己决定消息主体的格式，只要最后发送的 HTTP 请求满足上面的格式就可以。

但是，数据发送出去，还要服务端解析成功才有意义。一般服务端语言如 PHP、Python 等，以及它们的 framework，都内置了自动解析常见数据格式的功能。服务端通常是根据请求头（headers）中的 Content-Type 字段来获知请求中的消息主体是用何种方式编码，再对主体进行解析。所以说到 POST 提交数据方案，包含了 Content-Type 和消息主体编码方式两部分。下面就正式开始介绍它们:

- `application/x-www-form-urlencoded`

这是最常见的 POST 数据提交方式。浏览器的原生 `<form>` 表单，如果不设置 enctype 属性，那么最终就会以 `application/x-www-form-urlencoded` 方式提交数据。上个小节当中的例子便是使用了这种提交方式。可以看到 body 当中的内容和 GET 请求是完全相同的。

- `multipart/form-data`

这又是一个常见的 POST 数据提交的方式。我们使用表单上传文件时，必须让 `<form>` 表单的 enctype 等于 `multipart/form-data`。直接来看一个请求示例:

```
xxxxxxxxxx

POST http://www.example.com HTTP/1.1

Content-Type:multipart/form-data; boundary=----WebKitFormBoundaryrGKCBY7qhFd3TrwA


------WebKitFormBoundaryrGKCBY7qhFd3TrwA

Content-Disposition: form-data; name="text"


title

------WebKitFormBoundaryrGKCBY7qhFd3TrwA
```

```
Content-Disposition: form-data; name="file"; filename="chrome.png"

Content-Type: image/png


PNG ... content of chrome.png ...

------WebKitFormBoundaryrGKCBY7qhFd3TrwA--
```

这个例子稍微复杂点。首先生成了一个 boundary 用于分割不同的字段，为了避免与正文内容重复，boundary 很长很复杂。然后 `Content-Type` 里指明了数据是以 `multipart/form-data` 来编码，本次请求的 boundary 是什么内容。消息主体里按照字段个数又分为多个结构类似的部分，每部分都是以 --boundary 开始，紧接着是内容描述信息，然后是回车，最后是字段具体内容（文本或二进制）。如果传输的是文件，还要包含文件名和文件类型信息。消息主体最后以 --boundary-- 标示结束。关于 `multipart/form-data` 的详细定义，请前往 RFC1867 查看（或者相对友好一点的 MDN 文档）。

这种方式一般用来上传文件，各大服务端语言对它也有着良好的支持。

上面提到的这两种 POST 数据的方式，都是浏览器原生支持的，而且现阶段标准中原生 `<form>` 表单也只支持这两种方式（通过 `<form>` 元素的 enctype 属性指定，默认为 `application/x-www-form-urlencoded`。其实 enctype 还支持 text/plain，不过用得非常少）。

随着越来越多的 Web 站点，尤其是 WebApp，全部使用 Ajax 进行数据交互之后，我们完全可以定义新的数据提交方式，例如 `application/json`，`text/xml`，乃至 `application/x-protobuf` 这种二进制格式，只要服务器可以根据 `Content-Type` 和 `Content-Encoding` 正确地解析出请求，都是没有问题的。

响应报文

HTTP 响应与 HTTP 请求相似，HTTP响应也由3个部分构成，分别是：

- 状态行
- 响应头(Response Header)
- 响应正文

状态行由协议版本、数字形式的状态代码、及相应的状态描述，各元素之间以空格分隔。

常见的状态码有如下几种：

- `200 OK` 客户端请求成功
- `301 Moved Permanently` 请求永久重定向
- `302 Moved Temporarily` 请求临时重定向
- `304 Not Modified` 文件未修改，可以直接使用缓存的文件。
- `400 Bad Request` 由于客户端请求有语法错误，不能被服务器所理解。
- `401 Unauthorized` 请求未经授权。这个状态代码必须和WWW-Authenticate报头域一起使用
- `403 Forbidden` 服务器收到请求，但是拒绝提供服务。服务器通常会在响应正文中给出不提供服务的原因
- `404 Not Found` 请求的资源不存在，例如，输入了错误的URL
- `500 Internal Server Error` 服务器发生不可预期的错误，导致无法完成客户端的请求。
- `503 Service Unavailable` 服务器当前不能够处理客户端的请求，在一段时间之后，服务器可能会恢复正常。

下面是一个HTTP响应的例子：

```
xxxxxxxxxx

HTTP/1.1 200 OK
```

```
Server:Apache Tomcat/5.0.12

Date:Mon,6Oct2003 13:23:42 GMT

Content-Length:112


<html>...
```

条件 GET

HTTP 条件 GET 是 HTTP 协议为了减少不必要的带宽浪费，提出的一种方案。详见 RFC2616 。

1. HTTP 条件 GET 使用的时机?

   客户端之前已经访问过某网站，并打算再次访问该网站。

2. HTTP 条件 GET 使用的方法?

   客户端向服务器发送一个包询问是否在上一次访问网站的时间后是否更改了页面，如果服务器没有更新，显然不需要把整个网页传给客户端，客户端只要使用本地缓存即可，如果服务器对照客户端给出的时间已经更新了客户端请求的网页，则发送这个更新了的网页给用户。

   下面是一个具体的发送接受报文示例:

   客户端发送请求:

   ```
   xxxxxxxxxx
    GET / HTTP/1.1
    Host: www.sina.com.cn:80
    If-Modified-Since:Thu, 4 Feb 2010 20:39:13 GMT
    Connection: Close
   ```

   第一次请求时，服务器端返回请求数据，之后的请求，服务器根据请求中的 `If-Modified-Since` 字段判断响应文件没有更新，如果没有更新，服务器返回一个 `304 Not Modified`响应，告诉浏览器请求的资源在浏览器上没有更新，可以使用已缓存的上次获取的文件。

   ```
   xxxxxxxxxx
    HTTP/1.0 304 Not Modified
    Date: Thu, 04 Feb 2010 12:38:41 GMT
    Content-Type: text/html
    Expires: Thu, 04 Feb 2010 12:39:41 GMT
    Last-Modified: Thu, 04 Feb 2010 12:29:04 GMT
    Age: 28
    X-Cache: HIT from sy32-21.sina.com.cn
    Connection: close
   ```

   如果服务器端资源已经更新的话，就返回正常的响应。

持久连接

我们知道 HTTP 协议采用“请求-应答”模式，当使用普通模式，即非 Keep-Alive 模式时，每个请求/应答客户和服务器

都要新建一个连接，完成之后立即断开连接（HTTP 协议为无连接的协议）；当使用 Keep-Alive 模式（又称持久连接、连接重用）时，Keep-Alive 功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接。

在 HTTP 1.0 版本中，并没有官方的标准来规定 Keep-Alive 如何工作，因此实际上它是被附加到 HTTP 1.0协议上，如果客户端浏览器支持 Keep-Alive ，那么就在HTTP请求头中添加一个字段 Connection: Keep-Alive，当服务器收到附带有 Connection: Keep-Alive 的请求时，它也会在响应头中添加一个同样的字段来使用 Keep-Alive 。这样一来，客户端和服务器之间的HTTP连接就会被保持，不会断开（超过 Keep-Alive 规定的时间，意外断电等情况除外），当客户端发送另外一个请求时，就使用这条已经建立的连接。

在 HTTP 1.1 版本中，默认情况下所有连接都被保持，如果加入 "Connection: close" 才关闭。目前大部分浏览器都使用 HTTP 1.1 协议，也就是说默认都会发起 Keep-Alive 的连接请求了，所以是否能完成一个完整的 Keep-Alive 连接就看服务器设置情况。

由于 HTTP 1.0 没有官方的 Keep-Alive 规范，并且也已经基本被淘汰，以下讨论均是针对 HTTP 1.1 标准中的 Keep-Alive 展开的。

注意：

- HTTP Keep-Alive 简单说就是保持当前的TCP连接，避免了重新建立连接。
- HTTP 长连接不可能一直保持，例如 `Keep-Alive: timeout=5, max=100`，表示这个TCP通道可以保持5秒，max=100，表示这个长连接最多接收100次请求就断开。
- HTTP 是一个无状态协议，这意味着每个请求都是独立的，Keep-Alive 没能改变这个结果。另外，Keep-Alive也不能保证客户端和服务器之间的连接一定是活跃的，在 HTTP1.1 版本中也如此。唯一能保证的就是当连接被关闭时你能得到一个通知，所以不应该让程序依赖于 Keep-Alive 的保持连接特性，否则会有意想不到的后果。
- 使用长连接之后，客户端、服务端怎么知道本次传输结束呢？两部分：1. 判断传输数据是否达到了Content-Length 指示的大小；2. 动态生成的文件没有 Content-Length ，它是分块传输（chunked），这时候就要根据 chunked 编码来判断，chunked 编码的数据在最后有一个空 chunked 块，表明本次传输数据结束，详见这里。什么是 chunked 分块传输呢？下面我们就来介绍一下。

Transfer-Encoding

Transfer-Encoding 是一个用来标示 HTTP 报文传输格式的头部值。尽管这个取值理论上可以有很多，但是当前的 HTTP 规范里实际上只定义了一种传输取值——chunked。

如果一个HTTP消息（请求消息或应答消息）的Transfer-Encoding消息头的值为chunked，那么，消息体由数量未定的块组成，并以最后一个大小为0的块为结束。

每一个非空的块都以该块包含数据的字节数（字节数以十六进制表示）开始，跟随一个CRLF （回车及换行），然后是数据本身，最后块CRLF结束。在一些实现中，块大小和CRLF之间填充有白空格（0x20）。

最后一块是单行，由块大小（0），一些可选的填充白空格，以及CRLF。最后一块不再包含任何数据，但是可以发送可选的尾部，包括消息头字段。消息最后以CRLF结尾。

一个示例响应如下：

```
xxxxxxxxxx

HTTP/1.1 200 OK

Content-Type: text/plain

Transfer-Encoding: chunked
```

```
25

This is the data in the first chunk


1A

and this is the second one

0
```

注意：

- chunked 和 multipart 两个名词在意义上有类似的地方，不过在 HTTP 协议当中这两个概念则不是一个类别的。multipart 是一种 Content-Type，标示 HTTP 报文内容的类型，而 chunked 是一种传输格式，标示报头将以何种方式进行传输。
- chunked 传输不能事先知道内容的长度，只能靠最后的空 chunk 块来判断，因此对于下载请求来说，是没有办法实现进度的。在浏览器和下载工具中，偶尔我们也会看到有些文件是看不到下载进度的，即采用 chunked 方式进行下载。
- chunked 的优势在于，服务器端可以边生成内容边发送，无需事先生成全部的内容。HTTP/2 不支持 Transfer-Encoding: chunked，因为 HTTP/2 有自己的 streaming 传输方式（Source：MDN - Transfer-Encoding）。

HTTP Pipelining（HTTP 管线化）

默认情况下 HTTP 协议中每个传输层连接只能承载一个 HTTP 请求和响应，浏览器会在收到上一个请求的响应之后，再发送下一个请求。在使用持久连接的情况下，某个连接上消息的传递类似于`请求1 -> 响应1 -> 请求2 -> 响应2 -> 请求3 -> 响应3`。

HTTP Pipelining（管线化）是将多个 HTTP 请求整批提交的技术，在传送过程中不需等待服务端的回应。使用 HTTP Pipelining 技术之后，某个连接上的消息变成了类似这样`请求1 -> 请求2 -> 请求3 -> 响应1 -> 响应2 -> 响应3`。

注意下面几点：

- 管线化机制通过持久连接（persistent connection）完成，仅 HTTP/1.1 支持此技术（HTTP/1.0不支持）
- 只有 GET 和 HEAD 请求可以进行管线化，而 POST 则有所限制
- 初次创建连接时不应启动管线机制，因为对方（服务器）不一定支持 HTTP/1.1 版本的协议
- 管线化不会影响响应到来的顺序，如上面的例子所示，响应返回的顺序并未改变
- HTTP /1.1 要求服务器端支持管线化，但并不要求服务器端也对响应进行管线化处理，只是要求对于管线化的请求不失败即可
- 由于上面提到的服务器端问题，开启管线化很可能并不会带来大幅度的性能提升，而且很多服务器端和代理程序对管线化的支持并不好，因此现代浏览器如 Chrome 和 Firefox 默认并未开启管线化支持

更多关于 HTTP Pipelining 的知识可以参考这里。

会话跟踪

1. 什么是会话?

   客户端打开与服务器的连接发出请求到服务器响应客户端请求的全过程称之为会话。

2. 什么是会话跟踪?

   会话跟踪指的是对同一个用户对服务器的连续的请求和接受响应的监视。

3. 为什么需要会话跟踪?

浏览器与服务器之间的通信是通过HTTP协议进行通信的，而HTTP协议是”无状态”的协议，它不能保存客户的信息，即一次响应完成之后连接就断开了，下一次的请求需要重新连接，这样就需要判断是否是同一个用户，所以才有会话跟踪技术来实现这种要求。

1. 会话跟踪常用的方法:

    1. URL 重写

        URL(统一资源定位符)是Web上特定页面的地址，URL重写的技术就是在URL结尾添加一个附加数据以标识该会话,把会话ID通过URL的信息传递过去，以便在服务器端进行识别不同的用户。

    2. 隐藏表单域

        将会话ID添加到HTML表单元素中提交到服务器，此表单元素并不在客户端显示

    3. Cookie

        Cookie 是Web 服务器发送给客户端的一小段信息，客户端请求时可以读取该信息发送到服务器端，进而进行用户的识别。对于客户端的每次请求，服务器都会将 Cookie 发送到客户端,在客户端可以进行保存,以便下次使用。

        客户端可以采用两种方式来保存这个 Cookie 对象，一种方式是保存在客户端内存中，称为临时 Cookie，浏览器关闭后这个 Cookie 对象将消失。另外一种方式是保存在客户机的磁盘上，称为永久 Cookie。以后客户端只要访问该网站，就会将这个 Cookie 再次发送到服务器上，前提是这个 Cookie 在有效期内，这样就实现了对客户的跟踪。

        Cookie 是可以被客户端禁用的。

    4. Session:

        每一个用户都有一个不同的 session，各个用户之间是不能共享的，是每个用户所独享的，在 session 中可以存放信息。

        在服务器端会创建一个 session 对象，产生一个 sessionID 来标识这个 session 对象，然后将这个 sessionID 放入到 Cookie 中发送到客户端，下一次访问时，sessionID 会发送到服务器，在服务器端进行识别不同的用户。

        Session 的实现依赖于 Cookie，如果 Cookie 被禁用，那么 session 也将失效。

跨站攻击

- CSRF（Cross-site request forgery，跨站请求伪造）

  CSRF(XSRF) 顾名思义，是伪造请求，冒充用户在站内的正常操作。

  例如，一论坛网站的发贴是通过 GET 请求访问，点击发贴之后 JS 把发贴内容拼接成目标 URL 并访问:

  ```
  xxxxxxxxxx

    http://example.com/bbs/create_post.php?title=标题&content=内容
  ```

  那么，我们只需要在论坛中发一帖，包含一链接:

  ```
  xxxxxxxxxx

    http://example.com/bbs/create_post.php?title=我是脑残&content=哈哈
  ```

只要有用户点击了这个链接，那么他们的帐户就会在不知情的情况下发布了这一帖子。可能这只是个恶作剧，但是既然发贴的请求可以伪造，那么删帖、转帐、改密码、发邮件全都可以伪造。

如何防范 **CSRF** 攻击？可以注意以下几点：

- 关键操作只接受 POST 请求
- 验证码

  CSRF 攻击的过程，往往是在用户不知情的情况下构造网络请求。所以如果使用验证码，那么每次操作都需要用户进行互动，从而简单有效的防御了CSRF攻击。

  但是如果你在一个网站作出任何举动都要输入验证码会严重影响用户体验，所以验证码一般只出现在特殊操作里面，或者在注册时候使用。

- 检测 Referer

  常见的互联网页面与页面之间是存在联系的，比如你在 `www.baidu.com` 应该是找不到通往`www.google.com` 的链接的，再比如你在论坛留言，那么不管你留言后重定向到哪里去了，之前的那个网址一定会包含留言的输入框，这个之前的网址就会保留在新页面头文件的 `Referer`中

  通过检查 `Referer` 的值，我们就可以判断这个请求是合法的还是非法的，但是问题出在服务器不是任何时候都能接受到 `Referer` 的值，所以 Referer Check 一般用于监控 CSRF 攻击的发生，而不用来抵御攻击。

- Token

  目前主流的做法是使用 Token 抵御 CSRF 攻击。下面通过分析 CSRF 攻击来理解为什么 Token 能够有效

  CSRF 攻击要成功的条件在于攻击者能够预测所有的参数从而构造出合法的请求。所以根据不可预测性原则，我们可以对参数进行加密从而防止 CSRF 攻击。

  另一个更通用的做法是保持原有参数不变，另外添加一个参数 Token，其值是随机的。这样攻击者因为不知道 Token 而无法构造出合法的请求进行攻击。

  Token 使用原则

  - Token 要足够随机————只有这样才算不可预测
  - Token 是一次性的，即每次请求成功后要更新Token————这样可以增加攻击难度，增加预测难度
  - Token 要注意保密性————敏感操作使用 post，防止 Token 出现在 URL 中

  注意：过滤用户输入的内容**不能**阻挡 csrf，我们需要做的是过滤请求的**来源**。

- XSS（Cross Site Scripting，跨站脚本攻击）

XSS 全称“跨站脚本”，是注入攻击的一种。其特点是不对服务器端造成任何伤害，而是通过一些正常的站内交互途径，例如发布评论，提交含有 JavaScript 的内容文本。这时服务器端如果没有过滤或转义掉这些脚本，作为内容发布到了页面上，其他用户访问这个页面的时候就会运行这些脚本。

运行预期之外的脚本带来的后果有很多中，可能只是简单的恶作剧——一个关不掉的窗口：

```
xxxxxxxxxx
  while (true) {
      alert("你关不掉我~");
  }
```

也可以是盗号或者其他未授权的操作。

XSS 是实现 CSRF 的诸多途径中的一条，但绝对不是唯一的一条。一般习惯上把通过 XSS 来实现的 CSRF 称为 XSRF。

如何防御 XSS 攻击?

理论上，所有可输入的地方没有对输入数据进行处理的话，都会存在 XSS 漏洞，漏洞的危害取决于攻击代码的威力，攻击代码也不局限于 script。防御 XSS 攻击最简单直接的方法，就是过滤用户的输入。

如果不需要用户输入 HTML，可以直接对用户的输入进行 HTML escape 。下面一小段脚本：

```
xxxxxxxxxx

  <script>window.location.href="http://www.baidu.com";</script>
```

经过 escape 之后就成了：

```
xxxxxxxxxx

  &lt;script&gt;window.location.href=&quot;http://www.baidu.com&quot;&lt;/script&gt;
```

它现在会像普通文本一样显示出来，变得无毒无害，不能执行了。

当我们需要用户输入 HTML 的时候，需要对用户输入的内容做更加小心细致的处理。仅仅粗暴地去掉 script 标签是没有用的，任何一个合法 HTML 标签都可以添加 onclick 一类的事件属性来执行 JavaScript。更好的方法可能是，将用户的输入使用 HTML 解析库进行解析，获取其中的数据。然后根据用户原有的标签属性，重新构建 HTML 元素树。构建的过程中，所有的标签、属性都只从白名单中拿取。

参考资料

- 浅谈HTTP中Get与Post的区别
- http请求与http响应详细解析
- HTTP 条件 Get (Conditional Get)
- HTTP中的长连接与短连接
- HTTP Keep-Alive模式
- 分块传输编码
- HTTP 管线化(HTTP pipelining)
- HTTP协议及其POST与GET操作差异 & C#中如何使用POST、GET等
- 四种常见的 POST 提交数据方式
- 会话跟踪
- 总结 XSS 与 CSRF 两种跨站攻击
- CSRF简单介绍与利用方法
- XSS攻击及防御
- 百度百科：HTTP

二. HTTP over SSL/TLS

HTTPS 基本过程

HTTPS 即 HTTP over TLS，是一种在加密信道进行 HTTP 内容传输的协议。

> TLS 的早期版本叫做 SSL。SSL 的 1.0, 2.0, 3.0 版本均已经被废弃，出于安全问题考虑广大浏览器也不再对老旧的 SSL 版本进行支持了，因此这里我们就统一使用 TLS 名称了。

TLS 的基本过程如下（取自 what-happens-when-zh_CN）：

- 客户端发送一个 `ClientHello` 消息到服务器端，消息中同时包含了它的 Transport Layer Security (TLS) 版本，可用的加密算法和压缩算法。
- 服务器端向客户端返回一个 `ServerHello` 消息，消息中包含了服务器端的 TLS 版本，服务器所选择的加密和压缩算法，以及数字证书认证机构（Certificate Authority，缩写 CA）签发的服务器公开证书，证书中包含了公钥。客户端会使用这个公钥加密接下来的握手过程，直到协商生成一个新的对称密钥。证书中还包含了该证书所应用的域名范围（Common Name，简称 CN），用于客户端验证身份。
- 客户端根据自己的信任 CA 列表，验证服务器端的证书是否可信。如果认为可信（具体的验证过程在下一节讲解），客户端会生成一串伪随机数，使用服务器的公钥加密它。这串随机数会被用于生成新的对称密钥
- 服务器端使用自己的私钥解密上面提到的随机数，然后使用这串随机数生成自己的对称主密钥
- 客户端发送一个 `Finished` 消息给服务器端，使用对称密钥加密这次通讯的一个散列值
- 服务器端生成自己的 hash 值，然后解密客户端发送来的信息，检查这两个值是否对应。如果对应，就向客户端发送一个 `Finished` 消息，也使用协商好的对称密钥加密
- 从现在开始，接下来整个 TLS 会话都使用对称秘钥进行加密，传输应用层（HTTP）内容

从上面的过程可以看到，TLS 的完整过程需要三个算法（协议），密钥交互算法，对称加密算法，和消息认证算法（TLS 的传输会使用 MAC(message authentication code) 进行完整性检查）。

我们以 Github 网站使用的 TLS 为例，使用浏览器可以看到它使用的加密为 `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`。其中密钥交互算法是 `ECDHE_RSA`，对称加密算法是 `AES_128_GCM`，消息认证（MAC）算法为 `SHA256`。

TLS 证书机制

HTTPS 过程中很重要的一个步骤，是服务器需要有 CA 颁发的证书，客户端根据自己的信任 CA 列表验证服务器的身份。现代浏览器中，证书验证的过程依赖于证书信任链。

所谓证书信任链，即一个证书要依靠上一级证书来证明自己是可信的，最顶层的证书被称为根证书，拥有根证书的机构被称为根 CA。

还是以 Github 为例，在浏览器中我们可以看到它的证书信任链如下：

```
xxxxxxxxxx

DigiCert High Assurance EV Root CA` -> `DigiCert SHA2 Extended Validation Server CA` -> `Github.com
```

从上到下即 Root CA -> 二级 CA -> 网站。

前面提到，证书当中包括 CN(Common Name)，浏览器在验证证书的同时，也会验证 CN 的正确性。即不光需要验证“这是一个合法的证书”，还需要验证“这是一个用于 Github.com 的证书”。

既然所有的信任，最终要落到根 CA 上，根证书本身又是怎么获得的呢？答案也很简单，根证书一般是操作系统自带的。不管是桌面系统 Windows，macOS 还是移动端系统 Android, iOS 都会内置一系列根证书。随着操作系统本身的升级，根证书也会随着升级进行更新。

对浏览器而已，浏览器当然也有选择信任某个根证书的权利。Chrome 浏览器一般是跟随系统根证书信任的。Firefox 浏览器通常是使用自带的一套证书信任机制，不受系统证书的影响。

在使用 `curl` 等工具时，我们还可以自行选择证书进行信任。

> 有权威的信任，最终都要落到一个单点信任，不管是 Root CA，还是微软，苹果，谷歌等操作系统厂商。

中间人攻击

HTTPS 的过程并不是密不透风的，HTTPS 有若干漏洞，给中间人攻击（Man In The Middle Attack，简称 MITM）提供了可能。

所谓中间人攻击，指攻击者与通讯的两端分别建立独立的联系，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。

SSL 剥离

SSL 剥离即阻止用户使用 HTTPS 访问网站。由于并不是所有网站都只支持 HTTPS，大部分网站会同时支持 HTTP 和 HTTPS 两种协议。用户在访问网站时，也可能会在地址栏中输入 `http://` 的地址，第一次的访问完全是明文的，这就给了攻击者可乘之机。通过攻击 DNS 响应，攻击者可以将自己变成中间人。

> DNS 作为基于 UDP 的协议是相当不安全的，为了保证 DNS 的安全可以使用 DNS over TCP 等机制，这里不赘述了。

HSTS

为了防止上面说的这种情况，一种叫做 HSTS 的技术被引入了。HSTS（HTTP Strict Transport Security）是用于强制浏览器使用 HTTPS 访问网站的一种机制。它的基本机制是在服务器返回的响应中，加上一个特殊的头部，指示浏览器对于此网站，强制使用 HTTPS 进行访问：

```
xxxxxxxxxx

Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
```

可以看到如果这个过期时间非常长，就是导致在很长一段时间内，浏览器都会强制使用 HTTPS 访问该网站。

HSTS 有一个很明显的缺点，是需要等待第一个服务器的影响中的头部才能生效，但如果第一次访问该网站就被攻击呢？为了解决这个问题，浏览器中会带上一些网站的域名，被称为 HSTS preload list。对于在这个 list 的网站来说，直接强制使用 HTTPS。

伪造证书攻击

HSTS 只解决了 SSL 剥离的问题，然而即使在全程使用 HTTPS 的情况下，我们仍然有可能被监听。

假设我们想访问 `www.google.com`，但我们的 DNS 服务器被攻击了，指向的 IP 地址并非 Google 的服务器，而是攻击者的 IP。当攻击者的服务器也有合法的证书的时候，我们的浏览器就会认为对方是 Google 服务器，从而信任对方。这样，攻击者便可以监听我们和谷歌之前的所有通信了。

可以看到攻击者有两步需要操作，第一步是需要攻击 DNS 服务器。第二步是攻击者自己的证书需要被用户信任，这一步对于用户来说是很难控制的，需要证书颁发机构能够控制自己不滥发证书。

> 2015 年 Google 称发现赛门铁克旗下的 Thawte 未经同意签发了众多域名的数千个证书，其中包括 Google 旗下的域名和不存在的域名。当年 12 月，Google 发布公告称 Chrome、Android 及其他 Google 产品将不再信任赛门铁克旗下的"Class 3 Public Primary CA"根证书。

> 2016 年 Mozilla 发现沃通 CA 存在严重的信任问题，例如偷签 `github.com` 的证书，故意倒填证书日期绕过浏览器对 SHA-1 证书的限制等，将停止信任 WoSign 和 StartCom 签发的新证书。

HPKP

HPKP 技术是为了解决伪造证书攻击而诞生的。

HPKP（Public Key Pinning Extension for HTTP）在 HSTS 上更进一步，HPKP 直接在返回头中存储服务器的公钥指纹信息，一旦发现指纹和实际接受到的公钥有差异，浏览器就可以认为正在被攻击：

```
xxxxxxxxxx
```

```
Public-Key-Pins: pin-sha256="base64=="; max-age=expireTime [; includeSubDomains][; report-uri="reportURI"]
```

和 HSTS 类似，HPKP 也依赖于服务器的头部返回，不能解决第一次访问的问题，浏览器本身也会内置一些 HPKP 列表。

> HPKP 技术仍然不能阻止第一次访问的攻击问题，部署和配置 HPKP 相当繁琐，一旦网站配置错误，就会导致网站证书验证失败，且在过期时间内无法有效恢复。HPKP 的机制也引来了一些安全性问题。Chrome 67 中废除了对 HPKP 的支持，在 Chrome 72 中 HPKP 被彻底移除。

三. tcp-udp