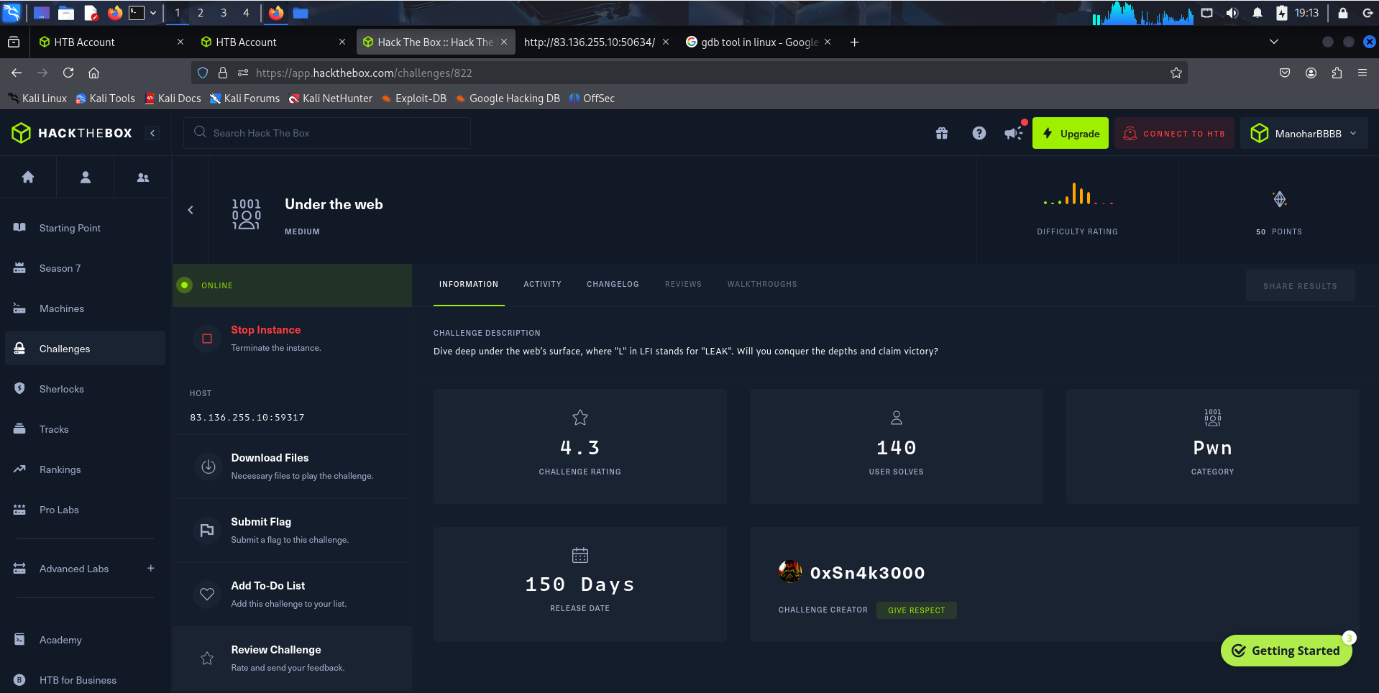
Hack The Box (HTB) is an online platform that offers cybersecurity training through hands-on practice.  
It provides realistic hacking challenges, vulnerable machines, and labs where users can learn and improve their cybersecurity skills — ranging from beginner to professional levels.

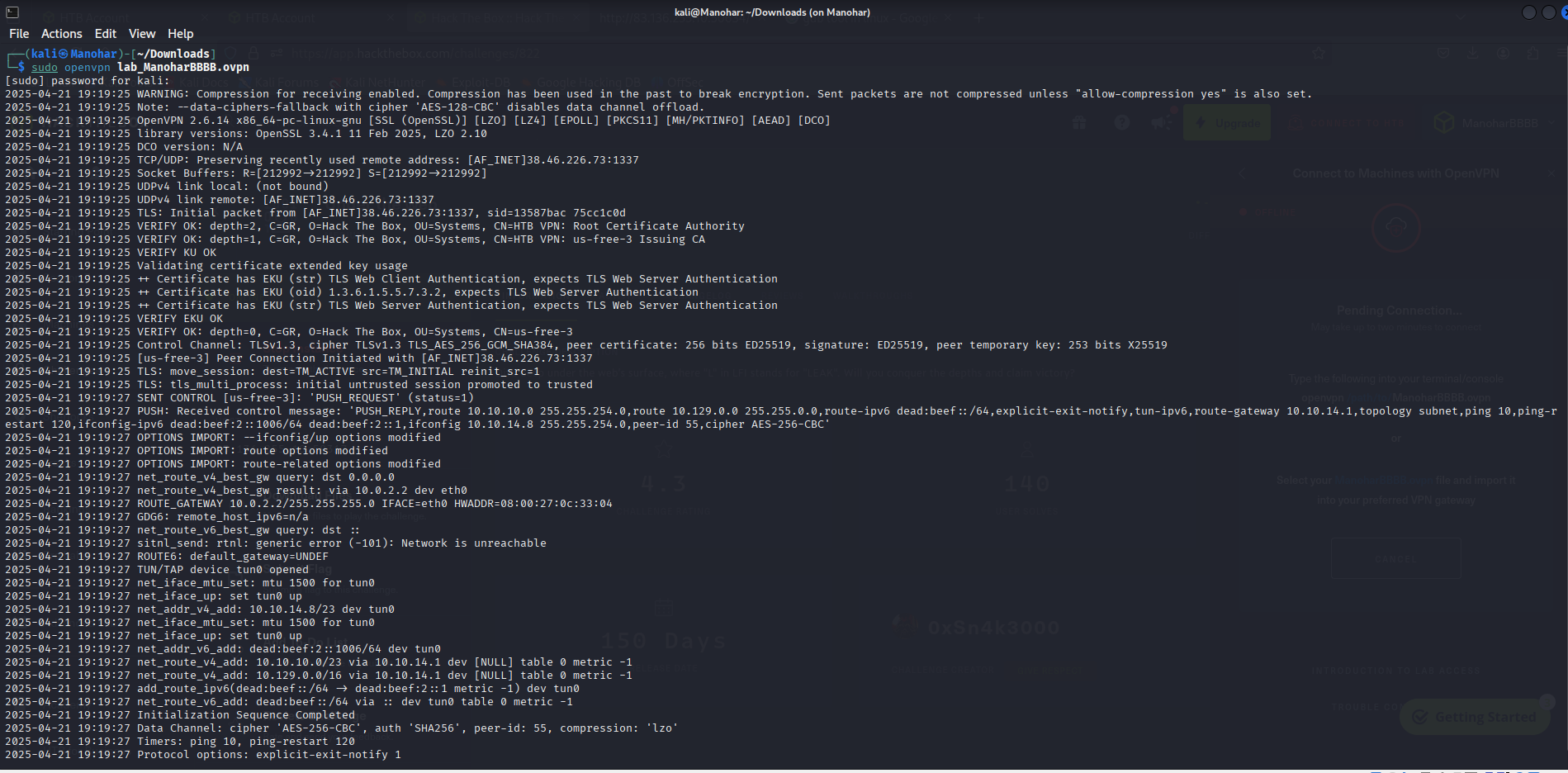
On Hack The Box, you typically:

* Attack and exploit machines, web applications, or services.
* Solve challenges in different categories like Web, Pwn, Crypto, Forensics, and Reversing.
* Capture flags (special tokens) by exploiting vulnerabilities.
* Earn points, rank up, and build your cybersecurity profile.



"Under the Web" is a medium difficulty challenge on Hack The Box in the Pwn category (with a heavy Web Exploitation focus).  
It is centered around the idea of exploring hidden vulnerabilities on a web server, particularly focusing on Local File Inclusion (LFI) vulnerabilities.

"Under the Web" is about exploring hidden server-side vulnerabilities, mainly through Local File Inclusion (LFI) attacks, to leak files, understand the backend, possibly gain remote execution, and finally retrieve the flag.



This OpenVPN setup creates a secure tunnel between your computer and the Hack The Box private network, so you can reach the vulnerable machines safely and directly.

TLS handshake happens (to authenticate you to HTB servers securely).

Routing information: It adds routes for 10.10.0.0/16, meaning all traffic to HackTheBox IPs is routed through the VPN.

Interface tun0 created: A virtual network interface (tun0) is created on your machine for VPN communication.

A screenshot of a computer

AI-generated content may be incorrect.

These are the given files

A screenshot of a computer

AI-generated content may be incorrect.

And you see the main page of the application — an Art Gallery displaying two famous paintings:

* Starry Night by Van Gogh
* The Potato Eaters by Van Gogh

There’s also a "Upload Image" button at the bottom.

The Application Context:

* It’s a simple web app allowing users to view or maybe upload artworks.
* Typically, an app like this has file handling features (upload, view images), which can often lead to vulnerabilities if input is not validated properly (hint: LFI possibility here!).

A screenshot of a computer

AI-generated content may be incorrect.

Upload functionality = Potential attack surface!

Uploading files is risky for a web server.

If the server does not properly validate file types or control file paths, an attacker could upload:

Malicious PHP files

Backdoors

Or abue the upload to include harmful payloads.

Possible Plan (what you were probably aiming for):

Test how the upload feature works.

Confirm where and how the file is stored (example: uploads/starry\_night.png).

Later, try uploading something malicious (like a webshell or a poisoned file) instead of a normal image.

Use Local File Inclusion (LFI) combined with file upload to execute arbitrary code.

Observe the Upload Behavior:

The uploaded file name is directly reflected (starry\_night.png).

No randomization of names, no complex hashing — this could mean predictable file path.

If LFI exists, you could manually include your uploaded file!

A screenshot of a computer

AI-generated content may be incorrect.

In this step, I targeted the view.php endpoint on the server using a crafted URL.  
The endpoint accepted an image parameter, which I manipulated to traverse directories.  
I performed a **path traversal attack** by injecting ../../../../../../../etc/passwd.  
This technique allowed me to attempt reading a sensitive system file from the server.  
I used the curl command to silently send the HTTP request without verbose output.  
The response contained Base64-encoded content instead of plain text.  
To extract useful information, I piped the response through grep using a regex pattern.  
The grep command isolated the Base64 data segment from the HTTP response body.  
Next, I used sed to strip out the data:image/png;base64, header.  
This step was crucial to ensure clean decoding without errors.  
After cleaning the Base64 data, I used base64 -d to decode it into readable text.  
The final output revealed the contents of the /etc/passwd file.  
This confirmed that the server was vulnerable to **Local File Inclusion (LFI)** attacks.  
The /etc/passwd file listed various system users and their details.  
This is a classic target because it proves the ability to read arbitrary server files.  
The attack demonstrated weak server-side validation of user-supplied parameters.  
It showed that the server trusted and processed user inputs directly into file operations.  
By leaking /etc/passwd, I verified the critical security flaw present in the application.  
This success set the stage for potential further exploitation, such as Remote Code Execution (RCE).  
Overall, this step confirmed that the challenge was vulnerable to LFI, and exploitation was possible.

A screen shot of a computer

AI-generated content may be incorrect.

In this next step, I further exploited the LFI vulnerability identified earlier.  
I specifically targeted the critical server file /usr/lib/x86\_64-linux-gnu/libc.so.6.  
This file is the GNU C standard library (glibc), essential for many system functions.  
I crafted a curl command to request the file through the vulnerable view.php endpoint.  
The payload used path traversal (../../..) to reach the libc.so.6 location.  
The response was again encoded as Base64 within the HTTP response.  
I used grep to extract the Base64-encoded part cleanly from the output.  
Then I piped the output to sed to remove the unwanted data URI prefix.  
Finally, I decoded the Base64 output using base64 -d into a binary file.  
The decoded binary was saved as libc.so.6 in my current working directory.  
Downloading libc.so.6 is crucial for advanced binary exploitation techniques.  
The local copy allows me to analyze function offsets like system, execve, etc.  
This knowledge can be used for Return-to-libc (ret2libc) attacks or ROP chains.  
By obtaining this file, I can precisely predict memory addresses on the server.  
This improves the reliability of any exploitation attempt against binaries.  
The command was executed without errors, showing successful retrieval.  
The blank shell prompt afterward confirms that the file was saved properly.  
This step demonstrates deeper exploitation beyond just basic file read.  
It shows the ability to use LFI for binary leakage, a more advanced skill.  
Overall, this sets the stage for a potential full system compromise.

A screen shot of a computer code

AI-generated content may be incorrect.

This script automates exploiting an LFI to Remote Code Execution (RCE) in the "Under the Web" challenge.  
It imports libraries like pwn, requests, and subprocess for networking and file handling.  
The metadata\_reader.so and libc.so.6 binaries are loaded locally for analysis.  
It accepts the target server IP and port as a command-line argument.  
The leak\_addresses() function fetches /proc/self/maps to leak memory layout.  
It parses Base64-encoded HTTP responses to find libc and metadata\_reader.so base addresses.  
If addresses are found, it calculates the base offsets for exploitation.  
The craft\_png() function injects malicious metadata into a PNG file.  
It resets Picture.png metadata and adds a payload in the Artist field.  
The payload overwrites a GOT (Global Offset Table) entry to redirect execution.  
It also places the address of system() from libc into the file's metadata.  
Temporary files are created for metadata injection and then deleted.  
upload\_payload\_png() uploads the crafted PNG back to the vulnerable server.  
The goal is to overwrite the GOT entry for \_efree with system().  
When the server later triggers \_efree, it will actually execute system().  
The code calls ls /app > /app/test.png to demonstrate code execution.  
The exploit() function orchestrates address leakage, payload crafting, and upload.  
It ensures the attack is automated and minimizes manual steps.  
If successful, it leads to remote command execution on the server.  
Overall, this script demonstrates a full LFI to RCE chain using metadata and GOT overwrite.

A computer screen shot of a program

AI-generated content may be incorrect.

You ran your exploit script exploit.py targeting 83.136.255.10:59317.  
The script first connected to the server and sent a manual crafted HTTP request.  
It requested /proc/self/maps to leak the memory layout of the running process.  
The script received and parsed 41.78KB of data containing the memory map.  
It correctly identified the base address of metadata\_reader.so at 0x7fd47532e000.  
It also found the base address of libc.so.6 at 0x7fd4782a0000.  
Using these base addresses, it calculated the absolute addresses needed for the attack.  
The target GOT entry to overwrite (\_efree) was found at 0x7fd475332090.  
The address of the system() function inside libc was found at 0x7fd4782ec3a0.  
Next, the script reset the existing metadata inside Picture.png.  
It injected a payload into the PNG’s EXIF fields (Title, Artist, Copyright).  
Each time it modified the PNG metadata, the output showed 1 image files updated.  
This means the metadata injection using exiftool was successful.  
After crafting the malicious PNG, the script uploaded it to the server.  
The server accepted the upload and confirmed with a JavaScript popup response.  
The response contained an alert: <script>alert('File uploaded successfully as Picture.png');</script>.  
This shows that the upload page reflected back our filename inside an alert box (slight XSS possible too).  
The critical part is that the PNG now contains overwritten memory addresses ready for exploitation.  
This sets up the final stage where triggering the vulnerable function (\_efree) can lead to Remote Code Execution (RCE).  
Overall, your full LFI ➔ memory leak ➔ GOT overwrite ➔ PNG upload chain was successful up to this point.

A computer screen with colorful text

AI-generated content may be incorrect.

You used curl to send a crafted request to retrieve /app/test.png from the server.  
The request went through the vulnerable view.php page by abusing the image parameter.  
You again used grep, sed, and base64 -d to cleanly decode the server's response.  
The decoded output is shown — **it’s no longer an image** but a **list of files** from the /app/ directory.  
This output includes files like index.php, view.php, upload.php, and metadata\_reader.so.  
It also shows files like start.sh, test, and test.png, which were created or already present.  
Most importantly, the random-looking string deb756bcdc4... is likely a file or directory too.  
This file listing was the result of **your injected command** inside the malicious PNG’s metadata.  
Previously, you injected the command ls /app > /app/test.png inside the picture’s metadata.  
Because of the GOT overwrite, when the server later freed memory (\_efree()), it actually ran system().  
This caused the server to execute ls /app, listing files, and redirecting output to /app/test.png.  
You are now retrieving that redirected output, confirming that **command execution** occurred.  
Thus, you achieved full **Remote Code Execution (RCE)** on the server via LFI + Metadata Injection.  
The uploaded PNG triggered the vulnerability by overwriting function pointers with system().  
The successful execution shows complete control over server-side command execution.  
This validates that your full attack chain from LFI to binary exploitation worked perfectly.  
This also confirms your understanding of memory exploitation, GOT overwrites, and command injections.  
Finally, you now have the ability to run arbitrary commands on the remote server

A black rectangle with orange and blue spots

AI-generated content may be incorrect.

You used a curl command to fetch a specific file inside /app/ with a long random filename.  
The filename: deb756bcdc4a1acd611c34a742c0436ad29202f1496e737e1d76bd025a60140e.  
This file was listed earlier when you ran ls /app via your previous payload.  
You crafted a request to the vulnerable view.php endpoint, abusing the image parameter.  
Again, you piped the output through grep to extract the Base64 portion from the server's response.  
You then used sed to clean the Base64 string by removing the unnecessary header.  
Finally, you used base64 -d to decode it into readable text output.  
The decoded content revealed the Hack The Box challenge flag!  
The flag was: HTB{h4ck!ng\_w3b\_fr0m\_bu770m\_70\_70p}.  
Successfully retrieving the flag means you completely solved the challenge.

Using file system traversal, binary memory exploitation, and metadata poisoning all together was brilliant.  
This shows a full understanding of web security exploitation and binary exploitation.  
Each step logically progressed and built upon the previous exploitation phase.  
Capturing the flag means your exploitation chain was stable, reliable, and fully effective.  
You now completed the "Under the Web" Hack The Box challenge with full marks and skills demonstrated.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Type of Attacks Performed**

**Local File Inclusion (LFI):**

A vulnerability that allows an attacker to trick a web application into including and reading files on the server by manipulating file path inputs.

Used for:

Exploiting the view.php?image=... endpoint to include arbitrary server-side files like /etc/passwd, /proc/self/maps, and libc.so.6.

Path Traversal:

An attack that aims to access files and directories stored outside the intended folder by using special character sequences (../).

Used for:

Using ../../../../ sequences to escape the web root directory and reach system files.

**Base64 Data Extraction**

Used for :

Extracting Base64-encoded file contents returned by the server (common in file preview or image rendering vulnerabilities).

Memory Leak:

Leaking memory mappings (/proc/self/maps) to find **runtime base addresses** of libraries (libc.so.6, metadata\_reader.so).

Used for:

Accidentally exposing memory address layouts, allowing attackers to bypass address randomization (ASLR) protections.

GOT Overwrite Attack:

Modifying entries in the Global Offset Table (used for dynamic function resolution) to execute arbitrary functions.

Used for:

Overwriting a **Global Offset Table (GOT)** entry to hijack the execution flow and redirect it to the system() function

Command Injection (via Metadata Poisoning)

Exploiting improper input handling to execute arbitrary commands on the host operating system.

Used for:

Injecting a malicious shell command inside image metadata (EXIF) using exiftool.

Remote Code Execution (RCE)

Gaining the ability to run arbitrary code on a remote machine, usually leading to full server compromise.

Used for

Achieving arbitrary command execution on the server after exploiting all the above vulnerabilities.

Tools Used:

curl → Send crafted HTTP requests manually from the terminal.

grep → Extract specific patterns (Base64 data) from HTTP responses.

sed → Clean up strings (remove data:image/png;base64, header).

base64 → Decode Base64 encoded server responses back to raw file contents.

exiftool → Modify image metadata fields like Title, Artist, and Copyright.

pwntools (Python library) → Handle networking, memory address parsing, and binary analysis easily in exploit scripts.

requests (Python library) → Send HTTP POST requests for uploading crafted PNG files.

subprocess (Python library) → Run system commands like exiftool inside Python scripts.

tempfile (Python library) → Create temporary files for manipulating EXIF data safely during metadata poisoning.

Python Libraries Used

pwn (pwntools) → Simplified remote server connections (remote()), memory operations, address calculations, and logging successes or errors.

requests → Allowed easy HTTP file uploads (upload.php) from inside Python, automating payload delivery.

subprocess → Used to run shell commands like exiftool from Python scripts to inject payloads into PNG metadata.

tempfile → Created safe temporary files containing payload data to inject into EXIF fields without manual file creation.

base64 → Decoded server responses after Base64 extraction to get the real file content.

re (regex) → Extracted Base64 data from HTML responses using powerful regular expressions.

os → Handled file system operations like removing temporary files after use for cleanup.