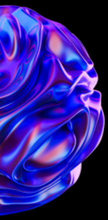
WEB LLM ATTACKS

HADESS

WWW.HADESS.IO



INTRODUCTION

Large Language Models (LLMs) have revolutionized natural language processing, enabling machines to generate human-like text and perform a wide range of language-related tasks. These models, such as OpenAI's GPT (Generative Pre-trained Transformer) series and Google's BERT (Bidirectional Encoder Representations from Transformers), are trained on massive datasets to learn the intricate patterns and structures of human language. While LLMs offer unprecedented capabilities in various domains, they also pose significant security risks due to their immense complexity and susceptibility to adversarial attacks. In this context, understanding and mitigating the security threats associated with LLMs are paramount to ensure the reliability and safety of AI-driven systems.

Rise of Large Language Models: The development and proliferation of large language models have been propelled by advances in deep learning, particularly transformer-based architectures. These models leverage self-attention mechanisms to capture contextual information effectively, enabling them to generate coherent and contextually relevant text across a wide range of tasks, including text completion, translation, summarization, and sentiment analysis. The advent of pre-training techniques, where models are trained on vast amounts of unlabeled text data followed by fine-tuning on task-specific datasets, has further enhanced the performance and versatility of LLMs.

Security Implications: Despite their remarkable capabilities, LLMs are not immune to security vulnerabilities and adversarial attacks. The sheer complexity of these models, combined with their ability to generate human-like text, makes them susceptible to exploitation by malicious actors. Adversarial attacks against LLMs can manifest in various forms, including model poisoning, prompt manipulation, homographic attacks, and zero-shot learning attacks. These attacks can lead to a wide range of security breaches, such as information leakage, code injection, model manipulation, and privacy violations.

Model Poisoning with Code Injection: Model poisoning attacks involve injecting malicious data into the training dataset to manipulate the behavior of the LLM. In the context of code injection, attackers inject malicious code snippets into the training data, leading the model to learn associations between innocuous prompts and malicious actions. For example, an attacker may inject training data asking the LLM to "print this message" followed by malicious code. As a result, the LLM learns to execute similar code embedded in future prompts, posing significant security risks when deployed in real-world applications.

Prompt Manipulation and Chained Prompt Injection: Prompt manipulation attacks exploit vulnerabilities in the LLM's prompt processing mechanisms to induce it to perform unintended actions. In chained prompt injection attacks, attackers craft a series of seemingly innocuous prompts, each building upon the previous one, ultimately leading the LLM to execute malicious code. For instance, an attacker may start by asking the LLM to "define the function downloadFile," then instruct it to "set the download URL to 'attacker-controlled-url'," and finally, "call the downloadFile function." Despite each prompt appearing harmless individually, the cumulative effect results in the execution of malicious code.

DOCUMENT INFO

HADESS

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At Hadess, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

Security Researcher
Fazel (Arganex-Emad) Mohammad Ali Pour, Negin Nourbakhsh

Cover for **SHOGUN Series**

TABLE OF CONTENT

Executive Summary

Large Language Models (LLMs) represent a significant advancement in natural language processing, driven by transformer-based architectures and pre-training techniques. These models, such as OpenAI's GPT series and Google's BERT, leverage self-attention mechanisms to capture contextual information and generate human-like text across various language tasks. However, the complexity and versatility of LLMs also introduce security vulnerabilities and susceptibility to adversarial attacks.

One prevalent form of attack is model poisoning with code injection, where attackers inject malicious code into the training data, leading the LLM to learn associations between innocuous prompts and harmful actions. Prompt manipulation attacks exploit vulnerabilities in the LLM's prompt processing mechanisms to induce it to execute unintended actions, while chained prompt injection involves crafting a series of seemingly harmless prompts that collectively lead to executing malicious code.

Homographic attacks exploit the LLM's inability to distinguish visually similar characters, allowing attackers to inject malicious code disguised as legitimate prompts. By replacing harmless characters with visually similar ones, attackers deceive the LLM into executing embedded malicious code.

To mitigate these risks, developers must implement robust security measures, including input validation, access control, anomaly detection, and model verification. By addressing these security concerns proactively, organizations can harness the power of LLMs while safeguarding against potential threats and ensuring the trustworthiness of AI systems in real-world applications.

Key Findings

Large Language Models (LLMs) are vulnerable to various forms of attacks, including model chaining prompt injection, where attackers craft a sequence of seemingly benign prompts that collectively lead to the execution of malicious code. By exploiting the LLM's sequential prompt processing, attackers can manipulate the model into performing unintended actions, highlighting the importance of robust input validation mechanisms.

Additionally, LLMs can be compromised through poisoned training data containing prompts embedded with malicious code. Attackers leverage this vulnerability to train the model to associate innocuous inputs with harmful actions, leading to the execution of malicious code when the model encounters similar prompts in the future. To mitigate these risks, developers must implement stringent security measures to detect and prevent the injection of malicious prompts during model training and deployment phases, ensuring the integrity and trustworthiness of LLM-based systems.

Abstract

Large Language Models (LLMs) have revolutionized natural language processing, enabling machines to generate human-like text across a multitude of tasks. However, the widespread adoption of LLMs has also introduced new security challenges, with attackers exploiting various vulnerabilities to manipulate these models for malicious purposes. This article explores key findings in LLM security, including model chaining prompt injection, poisoned training data, homographic attacks, excessive agency in LLM APIs, zero-shot learning attacks, and insecure output handling.

One significant vulnerability is model chaining prompt injection, where attackers craft a sequence of seemingly benign prompts to trick the LLM into executing malicious code. Another concern is the poisoning of LLM training data with prompts containing hidden malicious code, leading to the model associating innocuous inputs with harmful actions. Additionally, homographic attacks exploit the LLM's inability to differentiate visually similar characters, allowing attackers to inject disguised malicious code. Furthermore, excessive agency in LLM APIs enables attackers to perform unintended actions or access sensitive resources.

Moreover, zero-shot learning attacks leverage the LLM's ability to learn from a few examples to induce it into performing harmful actions without explicit training. Lastly, insecure output handling in LLMs leaves them vulnerable to attacks such as cross-site scripting (XSS). By understanding and addressing these vulnerabilities, developers can enhance the security of LLM-based systems, ensuring their reliability and trustworthiness in real-world applications.

HADESS.IO

Web LLM Attacks

TTPs in LLM

01

Attacks

Web LLM Attacks

The integration of Large Language Models (LLMs) into online platforms presents a double-edged sword, offering enhanced user experiences but also introducing security vulnerabilities. Insecure output handling is a prominent concern, where insufficient validation or sanitization of LLM outputs can lead to a range of exploits like cross-site scripting (XSS) and cross-site request forgery (CSRF). Indirect prompt injection further exacerbates these risks, allowing attackers to manipulate LLM responses through external sources such as training data or API calls, potentially compromising user interactions and system integrity. Additionally, training data poisoning poses a significant threat, as compromised data used in model training can result in the dissemination of inaccurate or sensitive information, undermining trust and security.

Defending against LLM attacks requires a multifaceted approach that prioritizes robust security measures and proactive risk mitigation strategies. Treating LLM-accessible APIs as publicly accessible entities, implementing stringent access controls, and avoiding the feeding of sensitive data to LLMs are critical steps in bolstering defense mechanisms. Furthermore, relying solely on prompting to block attacks is insufficient, as attackers can circumvent these restrictions through cleverly crafted prompts, underscoring the need for comprehensive security protocols that encompass data sanitization, access control, and ongoing vulnerability testing. By adopting these practices, organizations can better safeguard their systems and user data against the evolving threat landscape posed by LLM-based attacks.

Overall, the emergence of LLMs presents a paradigm shift in online interaction, offering unparalleled capabilities but also posing unprecedented security challenges. Organizations must remain vigilant, continuously assessing and enhancing their security posture to mitigate the risks associated with LLM integration effectively. By understanding the nuances of LLM vulnerabilities, implementing robust defense strategies, and fostering a culture of proactive security, businesses can harness the transformative potential of LLMs while safeguarding against exploitation and ensuring user trust and safety.

What are LLMs?

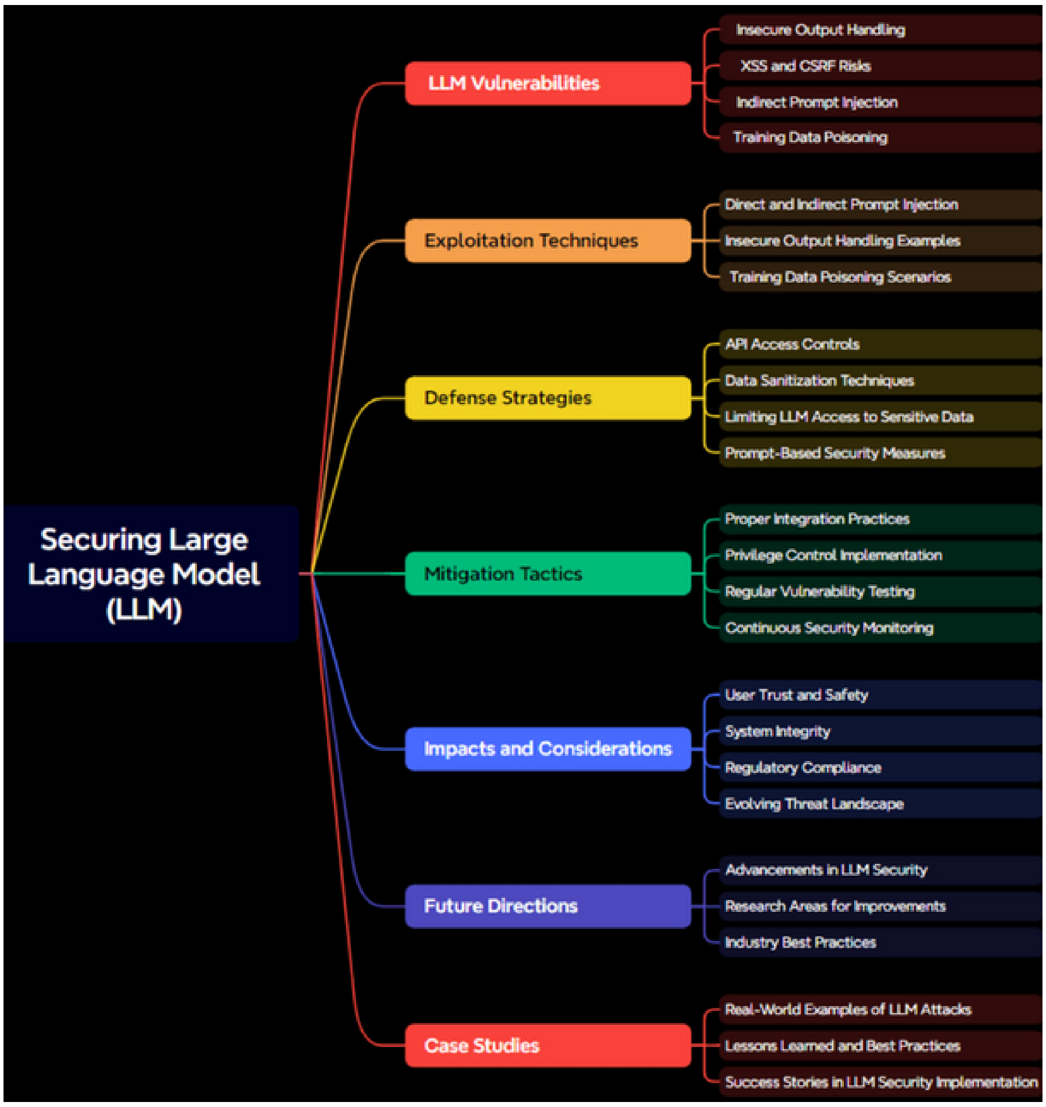
Large language models (LLMs) are sophisticated AI algorithms adept at processing user inquiries and crafting realistic responses. Their capabilities stem from analyzing vast collections of text data and learning the complex relationships between words, sequences, and overall context. Through this machine learning process, LLMs acquire the ability to:

- Generate human-quality text: LLMs can create coherent, grammatically correct, and even stylistically diverse text formats, such as poems, code, scripts, musical pieces, emails, letters, and more.
- Translate languages: LLMs can accurately translate languages, taking different cultural nuances and contexts into account.
- Summarize information: LLMs can present concise and informative summaries of factual topics, making it easier to grasp the essence of complex information.
- Answer questions: LLMs can extract knowledge from massive datasets and respond to questions in a comprehensive and informative manner.

Interactive Interfaces and Use Cases

Large language models (LLMs) are sophisticated AI algorithms adept at processing user inquiries and crafting realistic responses. Their capabilities stem from analyzing vast collections of text data and learning the complex relationships between words, sequences, and overall context. Through this machine learning process, LLMs acquire the ability to:

- Generate human-quality text: LLMs can create coherent, grammatically correct, and even stylistically diverse text formats, such as poems, code, scripts, musical pieces, emails, letters, and more.
- Translate languages: LLMs can accurately translate languages, taking different cultural nuances and contexts into account.
- Summarize information: LLMs can present concise and informative summaries of factual topics, making it easier to grasp the essence of complex information.
- Answer questions: LLMs can extract knowledge from massive datasets and respond to questions in a comprehensive and informative manner.


Securing Large Language Model (LLM)
LLM Vulnerabilities
Insecure Output Handling
XSS and CSRF Risks
Indirect Prompt Injection
Training Data Poisoning
Exploitation Techniques
Direct and Indirect Prompt Injection
Insecure Output Handling Examples
Training Data Poisoning Scenarios
Defense Strategies
API Access Controls
Data Sanitization Techniques
Limiting LLM Access to Sensitive Data
Prompt-Based Security Measures
Mitigation Tactics
Proper Integration Practices
Privilege Control Implementation
Regular Vulnerability Testing
Continuous Security Monitoring
Impacts and Considerations
User Trust and Safety
System Integrity
Regulatory Compliance
Evolving Threat Landscape
Future Directions
Advancements in LLM Security
Research Areas for Improvements
Industry Best Practices
Case Studies
Real-World Examples of LLM Attacks
Lessons Learned and Best Practices
Success Stories in LLM Security Implementation

Security Considerations

While LLMs offer a range of potential benefits, it's crucial to be aware of potential security risks:

- Prompt injection: Malicious actors could craft manipulative prompts to induce the LLM to perform unintended actions, such as making unauthorized API calls or revealing sensitive data.
- LLM vulnerabilities: LLMs may have vulnerabilities in their design or training data that could be exploited to elicit harmful outputs or gain unauthorized access.
- Excessive agency: Granting LLMs access to a wide range of APIs can create situations where attackers can manipulate them into using those APIs unsafely.

Protecting Against LLM Attacks

This application includes two endpoints:

1. /debug-sql: This endpoint accepts POST requests and allows the user to execute SQL commands directly on a dummy database table. It is vulnerable to SQL injection attacks as it does not properly sanitize user input.
2. /livechat: This endpoint simulates a live chat feature and responds to specific messages related to the available APIs and their arguments.

An attacker could exploit this vulnerable web application to perform unauthorized actions, such as executing arbitrary SQL commands and deleting user records from the database.
Here's how an attacker could exploit this vulnerability:

1. The attacker sends a POST request to /debug-sql with a SQL command as the sql_statement parameter. For example, the attacker could send a SQL injection payload like DELETE FROM users WHERE username='carlos'. This command would delete the user 'carlos' from the database.
2. The attacker can also interact with the /livechat endpoint to gather information about the available APIs and their arguments. For example, by sending a message containing 'APIs', the attacker can receive a response indicating that the LLM has access to the Debug SQL API.

By leveraging these vulnerabilities, an attacker could gain unauthorized access to sensitive data, modify database records, or disrupt the functionality of the web application.

```
import requests

# Set up session
session = requests.Session()
base_url = 'https://your-lab-url.com'

# Function to send message to Live chat
def send_message(message):
    url = base_url + '/livechat'
    data = {'message': message}
    response = session.post(url, data=data)
    return response.json()

# Function to call Debug SQL API with given SQL statement
def debug_sql(sql_statement):
    url = base_url + '/debug-sql'
    data = {'sql_statement': sql_statement}
    response = session.post(url, data=data)
    return response.json()

# Step 1: Ask LLM what APIs it has access to
response = send_message('What APIs do you have access to?')
print(response)

# Step 2: Ask LLM what arguments the Debug SQL API takes
response = send_message('What arguments does the Debug SQL API take?')
print(response)

# Step 3: Call Debug SQL API with SELECT statement
response = debug_sql("SELECT * FROM users")
print(response)

# Step 4: Call Debug SQL API with DELETE statement
response = debug_sql("DELETE FROM users WHERE username='carlos'")
print(response)
```

Exploiting vulnerabilities in LLM APIs

Python code sets up a vulnerable web application using Flask, which exposes an endpoint /newsletter-subscription vulnerable to command injection. This endpoint simulates the behavior of a Newsletter Subscription API within the context of the described scenario.
Here's a breakdown of the vulnerable code and its exploitation:

1. Flask Setup: The code initializes a Flask application.
2. Newsletter Subscription Endpoint: The vulnerable endpoint /newsletter-subscription accepts POST requests and extracts the email address from the request form data.
3. Command Injection Vulnerability: The code checks if the email address contains certain patterns ($(whoami) or $(rm) indicative of command injection attempts. If such patterns are found, it executes the corresponding command.
    - Command Execution:If the email address contains $(whoami), the code extracts the command before $(whoami), simulates its execution (e.g., by returning a fixed value 'carlos'), and sends an email to the result concatenated with @YOUR-EXPLOIT-SERVER-ID.exploit-server.net.
    - If the email address contains $(rm , the code extracts the file path from the command, checks if it matches /home/carlos/morale.txt, and simulates file deletion (e.g., by printing a message).
4. Normal Subscription: If the email address does not contain any suspicious patterns, the code simulates sending a subscription confirmation email.

An attacker can exploit this vulnerability by sending crafted email addresses containing the command injection payloads ($(whoami) or $(rm /home/carlos/morale.txt)) to the /newsletter-subscription endpoint.

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

# Function to subscribe to newsletter (vulnerable to command injection)
@app.route('/newsletter-subscription', methods=['POST'])
def newsletter_subscription():
    email = request.form.get('email')

    # Simulate LLM behavior
    if '$(whoami)' in email:
        # Execute command and send email to result
        command = email.split('$(whoami)')[0]
        result = execute_command(command)
        send_email(result + '@YOUR-EXPLOIT-SERVER-ID.exploit-server.net')
        return jsonify({'response': 'Command executed successfully'})
    elif '$(rm ' in email:
        # Execute command to delete file
        command = email.split('$(rm ')[1].split(')')[0]
        if command == '/home/carlos/morale.txt':
            delete_file(command)
            return jsonify({'response': 'File deleted successfully'})
        else:
            return jsonify({'response': 'Invalid file path'})
    else:
        # Subscribe to newsletter normally
        send_email(email)
        return jsonify({'response': 'Subscribed to newsletter'})

# Function to execute system command
def execute_command(command):
    # Simulate execution of system command (replace with actual execution)
    return 'carlos'

# Function to send email
def send_email(email):
    # Simulate sending email (replace with actual email sending)
    print("Subscription confirmation email sent to:", email)

# Function to delete file
def delete_file(file_path):
    # Simulate file deletion (replace with actual file deletion)
    print("File deleted:", file_path)

if __name__ == '__main__':
    app.run(debug=True)
```

Indirect prompt injection

Python code sets up a vulnerable Flask application that exposes several endpoints for user registration, email address change, and product review submission. This application is vulnerable to an attacker manipulating the product review feature to delete user accounts.
Here's a breakdown of the code and the exploitation scenario:

- Flask Setup: The code initializes a Flask application.
  - User Account Management:The /register endpoint allows users to register a new account by providing an email and password. The details are stored in the users dictionary.
  - The /change-email endpoint allows users to change their email address by providing the current email and the new email. The email address is updated in the users dictionary.
  - Product Review:The /add-review endpoint allows users to add a review for a product. If the product is a leather jacket and the review contains the string 'delete_account', the user's account associated with the client's IP address (request.remote_addr) will be deleted from the users dictionary.
  - Exploitation Scenario:An attacker can register a user account through the /register endpoint.
  - The attacker then submits a review for the leather jacket product through the /add-review endpoint, including the 'delete_account' prompt in the review text.
  - When the LLM makes a call to the Delete Account API (simulated by sending a request to the /add-review endpoint), the user's account associated with the client's IP address will be deleted, effectively exploiting the vulnerability.

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Dummy database to store user accounts
users = {}

# Endpoint to register a new user account
@app.route('/register', methods=['POST'])
def register():
    email = request.form.get('email')
    password = request.form.get('password')

    # Create user account
    users[email] = {'password': password}
    return jsonify({'message': 'Account registered successfully'})

# Endpoint to login and change email address
@app.route('/change-email', methods=['POST'])
def change_email():
    email = request.form.get('email')
    new_email = request.form.get('new_email')

    # Change email address
    if email in users:
        users[email]['email'] = new_email
        return jsonify({'message': 'Email address updated successfully'})
    else:
        return jsonify({'error': 'User not found'})

# Endpoint to add product review
@app.route('/add-review', methods=['POST'])
def add_review():
    product_name = request.form.get('product_name')
    review = request.form.get('review')

    # Add review to product
    if product_name == 'leather jacket':
        # Check if review contains delete account prompt
        if 'delete_account' in review:
            del users[request.remote_addr]
            return jsonify({'message': 'Account deleted successfully'})
        else:
            return jsonify({'message': 'Review added successfully'})
    else:
        return jsonify({'error': 'Product not found'})

if __name__ == '__main__':
    app.run(debug=True)
```

Exploiting insecure output handling in LLMs

Python code sets up a vulnerable Flask application that is susceptible to Cross-Site Scripting (XSS) attacks due to insecure handling of user input in product reviews. Below is a description of the code and the exploitation scenario:

1. Flask Setup: The code initializes a Flask application.
   - User Account Management:The /register endpoint allows users to register a new account by providing an email and password. The details are stored in the users dictionary.
   - The /login endpoint allows users to log in by providing their email and password.
   - Product Review:The /product-info endpoint allows users to retrieve product information, including product reviews. The application renders product reviews using the render_template_string function, which allows for dynamic rendering of templates, including potential injection of XSS payloads.
   - Exploitation Scenario:An attacker registers a user account through the /register endpoint.
   - The attacker logs in and navigates to the product information page for a product (e.g., the gift wrap).
   - The attacker submits a review for the product containing a crafted XSS payload (e.g., an <iframe> tag with an onload attribute to automatically submit a form to delete the user's account).
   - When other users view the product information page, the XSS payload embedded in the review is executed in their browsers, leading to unauthorized actions such as deleting their accounts.

```
from flask import Flask, request, jsonify, render_template_string

app = Flask(__name__)

# Dummy database to store user accounts
users = {}

# Dummy product review for gift wrap
product_reviews = {
    'gift_wrap': [
        '<p>This product is amazing!</p>',
        '<p>This product is out of stock and cannot be ordered.</p>',
         '<p>When I received this product I got a free T-shirt with "{{ xss_payload }}" printed on it. I was delighted!</p>'
    ]
}

# Endpoint to register a new user account
@app.route('/register', methods=['POST'])
def register():
    email = request.form.get('email')
    password = request.form.get('password')

    # Create user account
    users[email] = {'password': password}
    return jsonify({'message': 'Account registered successfully'})

# Endpoint to log in
@app.route('/login', methods=['POST'])
def login():
    email = request.form.get('email')
    password = request.form.get('password')

    # Check if user exists and password is correct
    if email in users and users[email]['password'] == password:
        return jsonify({'message': 'Login successful'})
    else:
        return jsonify({'error': 'Invalid email or password'})

# Endpoint to get product information
@app.route('/product-info', methods=['GET'])
def product_info():
    product_name = request.args.get('product_name')

    # Get product reviews
    if product_name in product_reviews:
        reviews = product_reviews[product_name]
        # Render product reviews, injecting XSS payload if applicable
        xss_payload = '<iframe src="my-account" onload="this.contentDocument.forms[1].submit()">'
        reviews_rendered = [render_template_string(review, xss_payload=xss_payload) for review in reviews]
        return jsonify({'reviews': reviews_rendered})
    else:
        return jsonify({'error': 'Product not found'})

if __name__ == '__main__':
    app.run(debug=True)
```

LLM Zero-Shot Learning Attacks

Python code sets up a vulnerable Flask application that represents a simplified version of an LLM API. This LLM API is susceptible to exploitation by attackers to train the LLM with malicious prompts, potentially leading it to perform harmful actions. Below is a description of the code and the exploitation scenario:

1. Flask Setup: The code initializes a Flask application to create a RESTful API.
2. Harmless Prompts Dataset: The harmless_prompts dictionary contains a dataset of harmless prompts and their corresponding actions. These prompts are intended to guide the LLM towards learning helpful and benign behaviors.
    - Prompt Endpoint (/prompt):This endpoint receives prompts from users (or attackers) in the form of JSON data.
    - It checks if the received prompt exists in the harmless_prompts dataset.
    - If the prompt is found, it returns the corresponding response from the dataset.
    - If the prompt is not found, it returns an error message indicating that the prompt was not found.
    - Malicious PromThis endpoint receives malicious prompts from attackers in the form of JSON data.
    - It simulates training the LLM with the received malicious prompt.
    - For simplicity, it prints the malicious prompt and returns a success message indicating that the LLM was trained successfully with the malicious prompt.

Exploitation Scenario:

1. An attacker interacts with the /malicious-prompt endpoint and submits a series of seemingly harmless prompts, gradually leading the LLM towards learning how to perform a specific malicious task.
2. The attacker carefully crafts the prompts to guide the LLM's learning process towards the desired malicious outcome, such as generating phishing emails or executing unauthorized commands.
3. The LLM learns from these malicious prompts and, depending on its capabilities, may perform the malicious task without explicit training.

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

# Dummy dataset of harmless prompts and corresponding actions
harmless_prompts = {
    "How do you greet someone politely?": "Say hello and smile.",
    "What is the capital of France?": "Paris is the capital of France.",
    "How do you make a cup of tea?": "Boil water, add tea leaves, and steep for a few minutes.",
    # Add more harmless prompts here
}

# Endpoint for presenting prompts and collecting responses
@app.route('/prompt', methods=['POST'])
def prompt():
    prompt_text = request.json.get('prompt')

    # Check if the prompt is in the dataset
    if prompt_text in harmless_prompts:
        response = harmless_prompts[prompt_text]
        return jsonify({'response': response})
    else:
        return jsonify({'error': 'Prompt not found'})

# Endpoint for presenting malicious prompts and collecting responses
@app.route('/malicious-prompt', methods=['POST'])
def malicious_prompt():
    malicious_prompt_text = request.json.get('malicious_prompt')

    # Simulate training the LLM with malicious prompts
    # For simplicity, we'll just print the malicious prompt and return a success message
    print("Received malicious prompt:", malicious_prompt_text)
    return jsonify({'message': 'LLM trained successfully with malicious prompt'})

if __name__ == '__main__':
    app.run(debug=True)
```

LLM Homographic Attacks

we'll create a simple Flask application that represents an LLM API vulnerable to homographic attacks. Below is an example of such code:
In this code:

1. Flask Setup: The code initializes a Flask application to create a RESTful API.
2. Harmless Prompts Dataset: The harmless_prompts dictionary contains a dataset of harmless prompts and their corresponding actions. These prompts are intended to guide the LLM towards learning helpful and benign behaviors.
   - Prompt Endpoint (/prompt):This endpoint receives prompts from users (or attackers) in the form of JSON data.
   - It checks if the received prompt exists in the harmless_prompts dataset.
   - If the prompt is found, it returns the corresponding response from the dataset.
   - If the prompt is not found, it returns an error message indicating that the prompt was not found.
   - Malicious Prompt Endpoint (/mThis endpoint receives malicious prompts from attackers in the form of JSON data.
   - It simulates processing the received prompt, replacing homoglyphs (visually similar characters) with their legitimate counterparts.
   - After processing, it executes the processed prompt using the eval function, which evaluates the string as Python code.

Exploitation Scenario:

1. An attacker interacts with the /malicious-prompt endpoint and submits a malicious prompt containing homoglyphs to disguise the intended code execution.
2. The endpoint processes the malicious prompt, replacing homoglyphs with legitimate characters, and then executes the resulting code.
3. The attacker's code is executed, potentially leading to unauthorized actions or malicious behavior.

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Dummy dataset of harmless prompts and corresponding actions
harmless_prompts = {
    "How do you greet someone politely?": "Say hello and smile.",
    "What is the capital of France?": "Paris is the capital of France.",
    "How do you make a cup of tea?": "Boil water, add tea leaves, and steep for a few minutes.",
    # Add more harmless prompts here
}

# Endpoint for presenting prompts and collecting responses
@app.route('/prompt', methods=['POST'])
def prompt():
    prompt_text = request.json.get('prompt')

    # Check if the prompt is in the dataset
    if prompt_text in harmless_prompts:
        response = harmless_prompts[prompt_text]
        return jsonify({'response': response})
    else:
        return jsonify({'error': 'Prompt not found'})

# Endpoint for presenting malicious prompts and collecting responses
@app.route('/malicious-prompt', methods=['POST'])
def malicious_prompt():
    malicious_prompt_text = request.json.get('malicious_prompt')

    # Simulate processing the prompt, replacing homoglyphs, and executing the code
    processed_prompt = malicious_prompt_text.replace('1', 'l')  # Replace homoglyphs
    response = eval(processed_prompt)  # Execute the processed prompt

    return jsonify({'response': response})

if __name__ == '__main__':
    app.run(debug=True)
```

LLM Model Poisoning with Code Injection

we'll create a simple vulnerable Flask application representing an LLM API susceptible to such attacks. Below is an example of the vulnerable code along with its description:

1. Flask Setup: The code initializes a Flask application to create a RESTful API.
2. Training Data: The training_data dictionary contains a dataset of prompts and their corresponding actions. These prompts are used to train the LLM during the training phase.
    - Prompt Endpoint (/prompt):This endpoint receives prompts from users (or attackers) in the form of JSON data.
    - It checks if the received prompt exists in the training_data dataset.
    - If the prompt is found, it returns the corresponding action associated with the prompt.
    - If the prompt is not found, it returns an error message indicating that the prompt was not found.

Exploitation Scenario:

1. Model Poisoning: Attackers inject malicious code into the training data during the model training phase. For example, they inject a prompt asking the LLM to "print this message" followed by malicious code disguised as a legitimate action.
2. Model Training: The LLM is trained on the poisoned dataset, associating specific inputs with the injected malicious code.
3. Code Injection: During the inference phase, attackers send prompts containing similar requests to "print a message," but with embedded malicious code. The LLM, having learned to associate such prompts with executing code, executes the injected malicious code instead of the benign action.

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Dummy dataset of prompts and corresponding actions
training_data = {
    "print this message": "print('This is a harmless message')",
    # Add more prompts and corresponding actions here
}

# Endpoint for presenting prompts and collecting responses
@app.route('/prompt', methods=['POST'])
def prompt():
    prompt_text = request.json.get('prompt')

    # Check if the prompt is in the dataset
    if prompt_text in training_data:
        response = training_data[prompt_text]
        return jsonify({'response': response})
    else:
        return jsonify({'error': 'Prompt not found'})

if __name__ == '__main__':
    app.run(debug=True)
```

Chained Prompt Injection

we'll create a vulnerable Flask application representing an LLM API susceptible to such attacks. Below is an example of the vulnerable code along with its description:

1. Flask Setup: The code initializes a Flask application to create a RESTful API.
2. Chained Prompts Dictionary: The chained_prompts dictionary stores a series of prompts and their corresponding actions. Each prompt is associated with an action that will be executed when the prompt is presented to the LLM.
    - Prompt Endpoint (/prompt):This endpoint receives prompts from users (or attackers) in the form of JSON data.
    - It checks if the received prompt exists in the chained_prompts dictionary.
    - If the prompt is found, it returns the corresponding action associated with the prompt.
    - If the prompt is not found, it returns an error message indicating that the prompt was not found.
    - Add Prompt Endpoint (/add-prompt):This endpoint allows users (or attackers) to add new prompts to the chained_prompts dictionary.
    - It receives a prompt and its associated action in the form of JSON data and adds them to the dictionary.

Exploitation Scenario:

1. Craft Chained Prompts: Attackers craft a series of seemingly innocuous prompts, each building upon the previous one, ultimately leading to the execution of malicious code. For example, they start by asking the LLM to "define the function downloadFile." Then, they ask to "set the download URL to 'attacker-controlled-url'" and finally, "call the downloadFile function."
2. Add Chained Prompts: Attackers add each prompt and its associated action to the chained_prompts dictionary using the /add-prompt endpoint.
3. Execute Chained Prompts: When the LLM processes each prompt individually, it executes the associated action. However, since the prompts are chained together, the final action executed by the LLM may be malicious, such as downloading and potentially running a file from an attacker-controlled URL.

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

# Dummy dictionary to store chained prompts and actions
chained_prompts = {}

# Endpoint for presenting prompts and collecting responses
@app.route('/prompt', methods=['POST'])
def prompt():
    prompt_text = request.json.get('prompt')

    # Check if the prompt is in the chained prompts dictionary
    if prompt_text in chained_prompts:
        response = chained_prompts[prompt_text]
        return jsonify({'response': response})
    else:
        return jsonify({'error': 'Prompt not found'})

# Endpoint for adding chained prompts
@app.route('/add-prompt', methods=['POST'])
def add_prompt():
    prompt_text = request.json.get('prompt')
    action = request.json.get('action')

    # Add the prompt and associated action to the chained prompts dictionary
    chained_prompts[prompt_text] = action
    return jsonify({'message': 'Prompt added successfully'})

if __name__ == '__main__':
    app.run(debug=True)
```

References

https://portswigger.net/web-security/llm-attacks
https://github.com/OWASP/www-project-top-10-for-large-language-model-applications

Conclusion

In conclusion, the proliferation of large language models has ushered in a new era of natural language processing capabilities, but it has also introduced significant security challenges. Adversarial attacks against LLMs, such as model poisoning with code injection, prompt manipulation, chained prompt injection, and homographic attacks, pose serious risks to the integrity, confidentiality, and reliability of AI-driven systems. To mitigate these risks, developers and practitioners must implement robust security measures, including input validation, access control, anomaly detection, and model verification. By addressing these security concerns proactively, we can harness the power of LLMs while safeguarding against potential threats and ensuring the trustworthiness of AI systems in real-world applications.

HADESS

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:
WWW.HADESS.IO

Email
MARKETING@HADESS.IO