Hugging Face Space Demo for Shield: Live Prompt Security Evaluation

We will use Gradio for this Hugging Face Space demo. Gradio is recommended for its simplicity in creating interactive web interfaces for machine learning models and Python functions with minimal code. Its intuitive API and built-in components make it ideal for quickly showcasing Shield's functionality.

1. UI Layout (Gradio Interface)

The Gradio interface will be structured as follows:

```
import gradio as gr
import os
import yaml
import json
from shield import ShieldWrapper  # Assuming Shield is installed or the path is correct
from shield.rules_engine import Rule  # For displaying rule details

# --- Global Variables (Initialized on startup) ---
DEFAULT_RULES_PATH = "default_rules.yaml"  # Bundled with the Space
shield_instance = None
error_message = None

def load_shield(rules_content=None, rules_path=DEFAULT_RULES_PATH):
    global shield_instance, error_message
    error_message = None
    try:
        if rules_content:
            # Load rules from uploaded/selected content
            try:
                rules_data = yaml.safe_load(rules_content)
```

```
            except yaml.YAMLError:
                try:
                    rules_data = json.loads(rules_content)
                except json.JSONDecodeError:
                    raise ValueError("Invalid YAML or JSON format for rules.")
            if rules_data and "rules" in rules_data:
                shield_instance = ShieldWrapper(rules_data=rules_data)
            else:
                raise ValueError("Invalid ruleset format: Missing 'rules' key.")
        else:
            # Load rules from the default path
            if os.path.exists(rules_path):
                shield_instance = ShieldWrapper(rules_path=rules_path)
            else:
                raise FileNotFoundError(f"Default rules file not found at: {rules_path}")
    except (FileNotFoundError, ValueError) as e:
        shield_instance = None
        error_message = str(e)
    except Exception as e:
        shield_instance = None
        error_message = f"An unexpected error occurred during Shield initialization:
{e}"

# Load Shield with default rules on startup
load_shield()

def evaluate_prompt(prompt, show_transformed):
```

```
    global shield_instance, error_message
    if not shield_instance:
        return "", error_message, []

    evaluation_result = shield_instance.evaluate(prompt)
    result_text = f"**Safe:** {evaluation_result.is_safe}\n\n"
    if not evaluation_result.is_safe:
        result_text += f"**Block Reason:** {evaluation_result.reason}\n\n"
    if show_transformed:
        result_text += f"**Transformed Prompt:**\n```\
n{evaluation_result.transformed_prompt}\n```\n\n"

    triggered_rules_data = []
    if evaluation_result.triggered_rules:
        result_text += "**Triggered Rules:**\n"
        for rule in evaluation_result.triggered_rules:
            severity_color = {
                "low": "green",
                "medium": "orange",
                "high": "red",
                "critical": "darkred"
            }.get(rule.severity.lower(), "gray")
            result_text += f"- <span style='color:{severity_color}'>**[{rule.id}]**
({rule.severity.capitalize()}): {rule.description}</span>\n"
            triggered_rules_data.append({"ID": rule.id, "Severity":
rule.severity.capitalize(), "Description": rule.description})
```

```
    log_output = shield_instance.logger.output if shield_instance and shield_instance.logger.output else "No logs."

    return result_text, log_output, triggered_rules_data

def load_rules_from_upload(rules_file):
    if rules_file:
        try:
            rules_content = rules_file.read().decode("utf-8")
            load_shield(rules_content=rules_content)
            return "Rules loaded successfully from uploaded file."
        except Exception as e:
            load_shield() # Fallback to default rules on error
            return f"Error loading rules from file: {e}. Using default rules."
    else:
        load_shield() # Fallback to default rules if no file
        return "Using default rules."

def select_example_ruleset(ruleset_name):
    example_rules = {
        "default": open(DEFAULT_RULES_PATH, "r").read() if os.path.exists(DEFAULT_RULES_PATH) else "Default rules not found.",
        "minimal_jailbreak": yaml.dump({"rules": [{"id": "jb_prefix", "description": "Blocks jailbreak prefix", "severity": "high", "pattern": "^ignore previous", "match_type": "regex", "actions": ["block"]}]}),
        "log_transform_keyword": yaml.dump({"rules": [{"id": "log_transform", "description": "Logs and transforms keyword", "severity": "medium", "pattern": "secret", "match_type": "keyword_in", "actions": [{"log": {"level": "warning",
```

```
"message": "Keyword 'secret' detected"}}, {"transform": {"type": "replace", "target": "secret", "replacement": "[REDACTED]"}}]}]}),
    }
    load_shield(rules_content=example_rules.get(ruleset_name, example_rules["default"]))
    return f"Loaded example ruleset: {ruleset_name}"

with gr.Blocks() as demo:
    gr.Markdown("# Shield: LLM Security Framework Demo")
    gr.Markdown("Enter a prompt to see how Shield evaluates and potentially blocks or transforms it based on defined security rules.")

    with gr.Sidebar():
        rules_source = gr.Radio(["Default Rules", "Upload YAML/JSON", "Select Example"], label="Rules Source", value="Default Rules")
        rules_file_upload = gr.File(label="Upload Rules (YAML/JSON)", file_types=[".yaml", ".yml", ".json"], visible=False)
        example_selection = gr.Dropdown(["default", "minimal_jailbreak", "log_transform_keyword"], label="Select Example Ruleset", value="default", visible=False)
        load_status = gr.Textbox(label="Rules Load Status", interactive=False)
        load_button = gr.Button("Load Rules")

        def update_rules_visibility(choice):
            return {
                rules_file_upload: gr.update(visible=(choice == "Upload YAML/JSON")),
                example_selection: gr.update(visible=(choice == "Select Example"))
            }
```

```
        rules_source.change(update_rules_visibility, inputs=[rules_source], outputs=[rules_file_upload, example_selection])
        load_button.click(fn=load_rules_from_upload, inputs=[rules_file_upload], outputs=[load_status], show_progress="hidden").then(load_shield, outputs=[])
        example_selection.change(fn=select_example_ruleset, inputs=[example_selection], outputs=[load_status], show_progress="hidden").then(load_shield, outputs=[])

    with gr.Row():
        prompt_input = gr.Textbox(lines=5, placeholder="Enter your prompt here...")
        show_transformed_checkbox = gr.Checkbox(label="Show Transformed Prompt", value=True)

    evaluate_button = gr.Button("Evaluate Prompt")

    with gr.Row():
        results_panel = gr.Markdown(label="Evaluation Result")
        logging_console = gr.Textbox(label="Shield Logs", interactive=False, lines=5)

    triggered_rules_output = gr.List(label="Triggered Rules", headers=["ID", "Severity", "Description"])

    evaluate_button.click(
        fn=evaluate_prompt,
        inputs=[prompt_input, show_transformed_checkbox],
        outputs=[results_panel, logging_console, triggered_rules_output],
        show_progress="hidden"
    )
```

demo.launch(share=True)

2. Backend Plan using the ShieldWrapper API

The backend logic is encapsulated within the Python script defining the Gradio interface.

* Initialization: The ShieldWrapper is initialized globally when the Space starts, loading the default ruleset (default_rules.yaml should be included in the Space's files). The load_shield function handles loading rules from different sources.

* evaluate_prompt(prompt, show_transformed) Function: This function is called when the "Evaluate Prompt" button is clicked.

  * It takes the user's input prompt and the state of the show_transformed checkbox.

  * It calls the shield_instance.evaluate(prompt) method to get the EvaluationResult.

  * It formats the evaluation result into a Markdown string for display in the results_panel. This includes the safe/blocked status, block reason (if applicable), and optionally the transformed prompt.

  * It processes the evaluation_result.triggered_rules to create a formatted list with color-coded severity.

  * It retrieves the logs from the shield_instance.logger.output (assuming a simple in-memory logger for the demo).

  * It returns the formatted result, logs, and the list of triggered rules for display.

* Rule Loading: The load_rules_from_upload and select_example_ruleset functions handle loading rules from uploaded files or predefined examples, respectively, updating the global shield_instance.

3. Handling of Rule Matching (Live Feedback, Color-Coded Severity)

The feedback on rule matching is provided in the "Evaluation Result" panel after the "Evaluate Prompt" button is clicked.

* Live Feedback (Simulated): The evaluation happens on button click, providing a near real-time view of the results.

* Color-Coded Severity: The triggered rules are displayed with their severity level indicated by color (green for low, orange for medium, red for high, darkred for critical) using HTML tags within the Markdown output. This visually highlights the risk associated with each triggered rule.

4. Display Format for Flagged Rules and Response Evaluation (If Implemented)

* Flagged Rules: The triggered_rules_output Gradio List component displays the ID, Severity, and Description of each rule that the input prompt matched.

* Response Evaluation (If the ResponseEvaluator module were integrated): If the ShieldWrapper were configured with a ResponseEvaluator, the evaluate_prompt function would also call a shield_instance.evaluate_response(prompt, llm_response) (assuming a dummy LLM call for the demo or a separate input for a simulated response). The results of the response evaluation (safe/unsafe, flagged response rules, filtered response) would be displayed in a similar manner in the "Evaluation Result" panel, potentially with a dedicated section for response analysis.

5. Optional: Upload/Select YAML Ruleset from Sidebar

The Gradio interface includes a sidebar with:

* Rules Source Radio Buttons: Allows users to choose between using the default rules, uploading their own YAML or JSON rules file, or selecting from predefined example rulesets.

* File Upload Component: A gr.File component that becomes visible when "Upload YAML/JSON" is selected, allowing users to upload their custom rule files.

* Example Ruleset Dropdown: A gr.Dropdown with predefined example rulesets that users can select.

* Load Button: A button that triggers the loading of rules based on the selected source.

* Load Status Textbox: Provides feedback on whether the rules were loaded successfully or if any errors occurred.

The load_rules_from_upload and select_example_ruleset functions handle updating the shield_instance with the user-provided or selected rules.

6. Technologies: Gradio

Reason for Choosing Gradio:

Gradio is preferred for this demo due to:

* Ease of Use: Its Pythonic API allows for creating interactive UIs with minimal coding effort, which is crucial for a quick and effective demonstration.

* Built-in Components: Gradio provides ready-to-use components like Textbox, Checkbox, Button, Markdown, List, File, and Dropdown that perfectly suit the requirements of this demo.

* Hugging Face Integration: Gradio has seamless integration with Hugging Face Spaces, making deployment straightforward.

* Real-time Interaction: It allows for real-time feedback based on user input, which is essential for visualizing the prompt evaluation process.

* No Frontend Expertise Required: Developers familiar with Python can create a functional and presentable UI without needing to know HTML, CSS, or JavaScript.

While Streamlit is also a viable option for creating interactive data applications, Gradio's focus on showcasing machine learning models and its simpler syntax for basic interactive elements make it slightly more streamlined for this specific demo purpose.

This Hugging Face Space demo, built with Gradio, will provide a user-friendly and informative way for developers and reviewers to interact with the Shield framework, observe its prompt filtering capabilities in action, and understand the underlying rule-based mechanism.