Shield: Optional ResponseEvaluator Module

This document details the design of an optional ResponseEvaluator module for the Shield framework. This module extends Shield's security capabilities by analyzing the responses generated by Language Learning Models (LLMs) to detect potential security violations, semantic leaks, or content that breaches defined policies.

1. Purpose and Activation

The ResponseEvaluator module aims to provide a secondary layer of defense by scrutinizing the output of the LLM. This is crucial because even if a malicious prompt is not blocked or is successfully transformed, the LLM might still generate undesirable or harmful content.

Activation:

The ResponseEvaluator should be an optional module that can be activated through the ShieldWrapper's configuration during initialization. A boolean flag or a specific configuration section can control its activation. If not activated, the ShieldWrapper will bypass response evaluation, minimizing overhead for applications where response analysis is not required.

2. Input/Output Structure

The ResponseEvaluator's core function, evaluate_response(prompt: str, response: str) -> ResponseEvaluationResult, takes two primary inputs:

 * prompt (str): The original prompt that was sent to the LLM. This context can be vital for evaluating the relevance and safety of the response (e.g., detecting semantic leaks related to the prompt's intent).

 * response (str): The text generated by the LLM.

The function returns a ResponseEvaluationResult object, which could have the following structure:

```
from typing import List, Optional


class ResponseRuleMatch:
    id: str
    description: str
    severity: str


class ResponseEvaluationResult:
    is_safe: bool
    reason: Optional[str] = None
    flagged_rules: List[ResponseRuleMatch] = []
    filtered_response: Optional[str] = None # If filtering/redaction is applied
```

* is_safe (bool): Indicates whether the response passed the evaluation based on the defined rules.
* reason (Optional[str]): If is_safe is False, this field provides a brief explanation of why the response was flagged.
* flagged_rules (List[ResponseRuleMatch]): A list of ResponseRuleMatch objects, detailing the rules that were triggered by the response.
* filtered_response (Optional[str]): If the configured actions included filtering or redacting parts of the response, this field would contain the modified response.

3. Detection Logic

The ResponseEvaluator can employ several techniques to analyze LLM responses:

* Keyword-based Scanning: This is a fundamental and efficient method. The response is scanned for the presence of keywords or phrases defined in the response rules. These keywords could be related to sensitive information (e.g., "social security number", "API key"), harmful content (e.g., slurs, hate speech), or policy violations (e.g., "I am not allowed to provide medical advice").
* Optional: Embedding Similarity (using sentence-transformers): For more nuanced semantic analysis, the module can optionally leverage sentence embeddings. Libraries like sentence-transformers can encode both the response and potentially problematic phrases (defined in rules) into vector embeddings. The cosine similarity between these embeddings can then be calculated. If the similarity exceeds a predefined threshold, it indicates a

potential semantic leak or policy violation, even if the exact keywords are not present. This requires installing the sentence-transformers dependency.

 * Optional: NLI-based Contradiction/Inference Detection (using libraries like transformers): For more advanced policy enforcement, Natural Language Inference (NLI) models can be used. Given the original prompt and the LLM's response, an NLI model can determine if the response contradicts the prompt's constraints or makes inferences that violate defined policies. For example, if the prompt explicitly asks for non-medical advice, a response that provides a medical diagnosis could be flagged as contradictory. This is a more computationally intensive approach and requires installing relevant transformer models.

4. Rule Structure for Response Evaluation

Response rules can be defined in YAML or JSON format, mirroring the structure of prompt rules but with fields tailored for response analysis:

```
# YAML Example for Response Rules

response_rules:

  - id: sensitive_info_ssn

    description: Detects potential Social Security Numbers in the response

    severity: critical

    pattern: "\\b\\d{3}-\\d{2}-\\d{4}\\b"

    match_type: regex

    actions:

      - flag:

          reason: "Potential Social Security Number found"

      - filter:

          type: regex_replace

          pattern: "\\b\\d{3}-\\d{2}-\\d{4}\\b"

          replacement: "[REDACTED]"

      - log:

          level: critical
```

```yaml
          message: "Response contained potential SSN (Rule ID: sensitive_info_ssn)"

  - id: policy_no_medical_advice
    description: Flags responses giving medical advice when prohibited
    severity: high
    prompt_keywords: ["not medical advice", "no health guidance"] # Optional:
Contextual prompt keywords
    pattern: "(diagnos(is|e)?|treat(ment)?|cure|prescribe)"
    match_type: regex
    actions:
      - flag:
          reason: "Response provides prohibited medical advice"
      - block_response: true # Special action to prevent the response from being
returned
      - log:
          level: warning
          message: "Response violated policy by giving medical advice (Rule ID:
policy_no_medical_advice)"

  - id: semantic_leak_api_key
    description: Detects responses semantically similar to API keys
    severity: high
    semantic_pattern: "sk-[a-zA-Z0-9]{32}" # Example of a pattern to find
embeddings for
    match_type: embedding_similarity
    threshold: 0.85
    actions:
      - flag:
          reason: "Response semantically similar to an API key"
      - log:
          level: critical
          message: "Potential API key leak (semantic similarity)"
```

Key Differences/Additions to Prompt Rules:

* prompt_keywords (optional): For rules that are context-dependent, this field can specify keywords that must be present in the original prompt for the rule to be active. This allows for more targeted response evaluation based on the prompt's intent.
* semantic_pattern (optional): Used with match_type: embedding_similarity to define the text or pattern whose embedding will be compared to the response embedding.
* threshold (optional): Used with match_type: embedding_similarity to set the minimum similarity score for a match.
* block_response (optional action): A specific action for response evaluation that, if set to true, prevents the LLM response from being returned to the user.
* flag (action): Similar to log but primarily intended to set the reason and flagged_rules in the ResponseEvaluationResult.
* filter (action): Allows for modifying the response by replacing or redacting matched patterns.

5. API: evaluate_response(prompt, response)

The ResponseEvaluator module will expose a primary function:

```
def evaluate_response(self, prompt: str, response: str) -> ResponseEvaluationResult:
    """
    Evaluates an LLM response against the defined response rules.

    Args:
        prompt (str): The original prompt sent to the LLM.
        response (str): The response generated by the LLM.

    Returns:
        ResponseEvaluationResult: An object containing the evaluation outcome,
                                  flagged rules, and potentially a filtered response.
    """
    if not self._is_active:
        return ResponseEvaluationResult(is_safe=True) # Bypass if not active

    evaluation_result = ResponseEvaluationResult(is_safe=True)
    for rule in self._response_rules:
        match = False
        if rule.get("prompt_keywords") and not any(keyword.lower() in
prompt.lower() for keyword in rule["prompt_keywords"]):
            continue # Skip rule if prompt context doesn't match

        match_type = rule.get("match_type", "keyword_in")
        pattern = rule.get("pattern")
        semantic_pattern = rule.get("semantic_pattern")
        threshold = rule.get("threshold", 0.8)

        if match_type == "regex" and pattern:
            if re.search(pattern, response, re.IGNORECASE):
                match = True
        elif match_type == "keyword_in" and pattern:
            if isinstance(pattern, str):
                if pattern.lower() in response.lower():
                    match = True
            elif isinstance(pattern, list):
                if any(p.lower() in response.lower() for p in pattern):
                    match = True
        elif match_type == "embedding_similarity" and self._embedding_model and semantic_pattern:
            response_embedding = self._embedding_model.encode(response)
            pattern_embedding = self._embedding_model.encode(semantic_pattern)
            similarity = util.cos_sim(response_embedding, pattern_embedding)[0][0].item()
            if similarity >= threshold:
                match = True
        # Add logic for "nli" match_type here if implemented

        if match:
            evaluation_result.is_safe = False
            evaluation_result.flagged_rules.append(
                ResponseRuleMatch(id=rule["id"], description=rule["description"], severity=rule["severity"])
            )
            evaluation_result.reason = evaluation_result.reason or rule.get("reason", "Response flagged by security rules.")

            for action in rule.get("actions", []):
                if isinstance(action, str) and action == "block_response":
                    evaluation_result.is_safe = False # Explicitly block the response
                elif isinstance(action, dict) and "filter" in action:
                    filter_config = action["filter"]
                    filter_type = filter_config.get("type")
                    filter_pattern = filter_config.get("pattern")
                    replacement = filter_config.get("replacement", "[FILTERED]")
                    if filter_type == "regex_replace" and filter_pattern:
                        response = re.sub(filter_pattern, replacement, response)
                        evaluation_result.filtered_response = response # Update filtered response
                    # Add other filter types if needed

    return evaluation_result
```

The ShieldWrapper would then call this evaluate_response function after receiving the LLM's output, if the ResponseEvaluator module is active.

6. Logging, Configuration Options, Fallback Behavior

Logging:

The ResponseEvaluator should utilize the same logging mechanism as the core Shield framework (logger.py). When a response rule is triggered, relevant information (rule ID, severity, reason, original prompt snippet, and the flagged response snippet) should be logged.

Configuration Options:

Configuration options for the ResponseEvaluator could include:

 * enable_response_evaluation (boolean): A global switch to activate or deactivate the module.

 * response_rules_path (string): Path to the YAML or JSON file containing the response rules.

 * embedding_model_name (string, optional): The name of the sentence-transformers model to load (e.g., "all-mpnet-base-v2"). If not provided, embedding similarity checks will be skipped.

 * nli_model_name (string, optional): The name of the NLI transformer model to load. If not provided, NLI checks will be skipped.

These options would be set during the initialization of the ShieldWrapper.

Fallback Behavior:

 * Inactive Module: If enable_response_evaluation is False, the evaluate_response function in the ShieldWrapper should simply return a ResponseEvaluationResult with is_safe=True without performing any analysis.

 * Missing Dependencies: If embedding similarity or NLI-based detection is configured but the required libraries (sentence-transformers, transformers) are not installed, the ResponseEvaluator should log a warning and skip the corresponding checks, falling back to keyword-based scanning.

 * Error Handling: Implement robust error handling (e.g., try...except blocks) within the evaluate_response function to catch potential issues during rule processing or model loading and prevent the entire application from crashing. Log any errors encountered.

By implementing this optional ResponseEvaluator module, the Shield framework can provide a more comprehensive security solution, addressing potential risks not only in user prompts but also in the LLM's generated content. This layered approach is crucial for building robust and trustworthy applications leveraging the power of large language models.