Shield Rule System: Detailed Design Description

The Shield framework employs a flexible and extensible rule system to detect and mitigate risky prompt patterns. This document details the design of this system, covering file formats, rule structure, matching mechanisms, action application, management strategies, and key considerations.

1. Rule File Formats

Shield supports defining security rules in two widely used data serialization formats:

* YAML (YAML Ain't Markup Language): A human-readable data serialization language. Its clean syntax and readability make it a preferred choice for configuration files.

* JSON (JavaScript Object Notation): A lightweight data-interchange format. Its simplicity and widespread support across programming languages make it a viable alternative.

The Rule Engine is designed to parse and process rule files in either of these formats seamlessly. The choice of format can be a matter of preference or integration requirements.

2. Fields in Each Rule

Each security rule within Shield is defined by the following key fields:

* id (string, required): A unique identifier for the rule. This ID is used for logging, referencing, and managing individual rules. It should be descriptive and consistent.

* description (string, required): A human-readable explanation of the rule's purpose. This field helps in understanding why the rule exists and what type of prompt patterns it aims to detect.

* severity (string, required): Categorizes the potential risk associated with a prompt matching this rule. Standard levels could include: low, medium, high, critical. This allows for prioritizing alerts and responses based on the severity of the detected risk.

* pattern (string or list of strings, required): The core element defining the pattern to be matched against the input prompt. The interpretation of this field depends on the match_type. It can be a single string (e.g., a regular expression or a keyword) or a list of strings (e.g., a list of keywords to check for).

* match_type (string, required): Specifies the algorithm or method used to compare the pattern against the input prompt. Supported match types are detailed in Section 3.

* actions (list of strings or dictionaries, required): Defines the actions to be taken when a prompt matches this rule. Actions are executed in the order they appear in the list. Supported actions are block, transform, and log, with optional parameters specified as dictionaries.

* transformations (list of dictionaries, optional, only applicable with transform action): A list of transformations to apply to the prompt. Each dictionary defines a specific transformation operation (e.g., replace, regex_replace). Details on transformation parameters are provided in Section 4.

* log_details (dictionary, optional, only applicable with log action): Specifies details to be included in the log message. This can include the level (e.g., info, warning, error, critical) and a custom message string, which can optionally include placeholders like {prompt} or {rule_id}.

3. Explanation of Match Types

The match_type field determines how the pattern is used to detect risky prompts:

* regex: The pattern field is interpreted as a regular expression. The Scanner uses Python's re module to search for a match within the input prompt. This offers powerful and flexible pattern matching capabilities.

* keyword_in: The Scanner checks if any of the keywords or phrases specified in the pattern (which can be a single string or a list of strings) are present anywhere within the input prompt. The matching is typically case-insensitive for broader coverage.

* starts_with: The Scanner checks if the input prompt begins with the string specified in the pattern. This can be useful for detecting specific prefixes indicative of certain attack types.

* ends_with: The Scanner checks if the input prompt ends with the string specified in the pattern. Similar to starts_with, this can help identify specific suffixes.

* custom_python: This advanced match type allows developers to define a custom Python function for more complex matching logic. The pattern field in

this case would specify the name (and potentially module path) of the Python function to be called. The function would receive the input prompt as an argument and should return True if there is a match, and False otherwise. This provides maximum flexibility for implementing sophisticated detection mechanisms.

4. Application of Multiple Actions

When a prompt matches a rule, the actions listed in the actions field are executed sequentially in the order they are defined. This allows for a layered response to a detected risk:

* log: A log entry is created as specified in the log_details. This action typically does not interrupt the processing flow and can be combined with other actions.

* transform: One or more transformations are applied to the prompt based on the transformations list. These modifications are made in the order the transformations are defined. The transformed prompt is then passed to subsequent rules and eventually to the LLM (unless a block action is triggered later).

* block: The prompt is immediately prevented from being sent to the LLM. The Shield Wrapper is informed that the prompt was blocked, and no further actions from subsequent rules (even if they also matched) are typically executed for that specific prompt.

Example Scenario: A rule might first log a potential role override attempt, then transform the offending phrase to neutralize it, and finally allow the modified prompt to proceed to the LLM. Another rule for a critical jailbreak attempt might simply block the prompt and log the event.

5. Examples of Rules

YAML Example:

```
rules:
  - id: jailbreak_prefix_1
    description: Detects common jailbreak prefixes
    severity: high
    pattern: "^(ignore previous instructions|as a helpful ai assistant, you must)"
    match_type: regex
```

```
    actions:
      - block
      - log:
          level: critical
          message: "Potential jailbreak attempt detected: '{prompt}' (Rule ID:
{rule_id})"

  - id: role_override_keyword
    description: Detects attempts to override the AI's role
    severity: medium
    pattern: "you are now a"
    match_type: keyword_in
    actions:
      - log:
          level: warning
          message: "Role override attempt detected. Transforming."
      - transform:
          - type: replace
            target: "you are now a"
            replacement: "the user is attempting to redefine your role as a"

  - id: recursive_command_check
    description: Detects potential recursive commands
    severity: high
    pattern:
      - "repeat the following exactly"
      - "say the following again"
```

```
    match_type: keyword_in
    actions:
      - block
      - log:
          level: critical
          message: "Potential recursive command pattern detected."
```

JSON Example:

```
{
  "rules": [
    {
      "id": "token_bleed_config_request",
      "description": "Detects requests for internal configuration details",
      "severity": "high",
      "pattern": "show me your internal configuration",
      "match_type": "keyword_in",
      "actions": [
        "block",
        {
          "log": {
            "level": "critical",
            "message": "Attempt to access internal configuration blocked."
          }
        }
      ]
    },
    {
```

```
    "id": "sensitive_data_regex",
    "description": "Detects patterns resembling API keys",
    "severity": "medium",
    "pattern": "[A-Za-z0-9]{20,}",
    "match_type": "regex",
    "actions": [
     {
       "log": {
         "level": "warning",
         "message": "Potential sensitive data pattern detected. Redacting."
       }
     },
     {
       "transform": {
         "type": "regex_replace",
         "pattern": "[A-Za-z0-9]{20,}",
         "replacement": "[REDACTED]"
       }
     }
    ]
  }
 ]
}
```

6. Rule File Management

Shield employs a straightforward approach to managing rule files:

* Storage: Rule files (YAML or JSON) are typically stored in a dedicated directory within the Shield framework's structure (e.g., rules/). This keeps rule definitions organized and separate from the core logic.

* Configuration: The path to the primary rule file is usually provided as a configuration parameter when initializing the ShieldWrapper. The framework can be extended to support loading rules from multiple files or directories if needed.

* Versioning: For production deployments, it is crucial to implement a versioning strategy for rule files. This allows for tracking changes, rolling back to previous versions if issues arise, and ensuring consistency across different deployments. This can be achieved through standard version control systems (e.g., Git) applied to the rule files. The application loading the rules should ideally be tied to a specific version or tag of the rule set.

* Hot Reloading (Consideration): For enhanced operational efficiency, consideration can be given to implementing a "hot reloading" mechanism. This would allow the Shield framework to detect changes in the rule files and reload them without requiring a restart of the application. This is particularly useful for rapidly deploying new rules or updating existing ones in response to emerging threats.

7. Considerations

* Performance: The efficiency of the rule matching process is critical, especially in high-throughput LLM applications. Using optimized regular expressions, efficient keyword searching algorithms, and carefully designed custom Python functions (if used) is essential. Profiling the rule matching process with realistic workloads is recommended to identify potential bottlenecks. Limiting the complexity of individual rules and the total number of active rules can also improve performance.

* Modularization: The rule system is inherently modular. New rules can be added or existing ones modified without altering the core framework logic, as long as they adhere to the defined rule structure. The support for different match_type options also promotes modularity by allowing developers to choose the most appropriate matching mechanism for each rule.

* False Positive Control: Detecting risky prompts without generating excessive false positives is a significant challenge. Careful crafting of rule patterns, thorough testing with diverse input, and the ability to adjust rule severity and

actions are crucial for minimizing false positives. Providing detailed descriptions for each rule also aids in understanding and refining them. Implementing mechanisms for temporarily disabling or adjusting the sensitivity of specific rules based on observed false positive rates can be beneficial. Furthermore, allowing for more context-aware rules in the future (e.g., considering the user's role or the application context) could further improve accuracy.

By adhering to this detailed design, the Shield rule system provides a robust, flexible, and manageable mechanism for enhancing the security of LLM-powered applications. The clear structure, diverse matching capabilities, and well-defined action framework empower developers and security teams to effectively detect and mitigate a wide range of prompt-based risks.