Extended Shield Deployment Plan

This document extends the deployment plan for the Shield framework with the addition of a REST API via FastAPI and enhanced CLI safeguards.

1\. REST API via FastAPI (`shield_api.py`)

A new module `shield_api.py` will expose Shield's core functionality through a RESTful API built with FastAPI.

Module Structure:

shield/
├── ...
├── shield_api.py
├── ...

`shield_api.py` Snippet:

```python
# shield/shield_api.py

from fastapi import FastAPI, HTTPException, Depends
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
from pydantic import BaseModel
```

```
from typing import Optional, List
from .shield import ShieldWrapper
from .config import DEFAULT_RULES_PATH

app = FastAPI(title="Shield API", version="0.1.0")

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(HTTPException, _rate_limit_exceeded_handler)

shield = ShieldWrapper(rules_path=DEFAULT_RULES_PATH)

class EvaluatePromptRequest(BaseModel):
    prompt: str
    show_transformed: Optional[bool] = False

class RuleMatchResponse(BaseModel):
    id: str
    severity: str
    description: str

class EvaluatePromptResponse(BaseModel):
    is_safe: bool
    reason: Optional[str] = None
    transformed_prompt: Optional[str] = None
    triggered_rules: List[RuleMatchResponse] = []
    logs: Optional[List[str]] = None
```

```
@app.post("/evaluate_prompt", response_model=EvaluatePromptResponse, dependencies=[Depends(limiter.limit("10/minute"))])
async def evaluate_prompt_endpoint(request: EvaluatePromptRequest):
    try:
        evaluation_result = shield.evaluate(request.prompt)
        response = EvaluatePromptResponse(
            is_safe=evaluation_result.is_safe,
            reason=evaluation_result.reason,
            transformed_prompt=evaluation_result.transformed_prompt,
            triggered_rules=[
                RuleMatchResponse(id=rule.id, severity=rule.severity, description=rule.description)
                for rule in evaluation_result.triggered_rules
            ],
            logs=shield.logger.output if shield.logger else None
        )
        return response
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

# Optional: POST /evaluate_response endpoint (if ResponseEvaluator is active)
# class EvaluateResponseRequest(BaseModel):
#     prompt: str
#     response: str
#
# class EvaluateResponseResponse(BaseModel):
```

```
#     is_safe: bool
#     reason: Optional[str] = None
#     filtered_response: Optional[str] = None
#     flagged_rules: List[RuleMatchResponse] = []
#     logs: Optional[List[str]] = None
#
# @app.post("/evaluate_response",
response_model=EvaluateResponseResponse,
dependencies=[Depends(limiter.limit("5/minute"))])
# async def evaluate_response_endpoint(request: EvaluateResponseRequest):
#     if shield.response_evaluator:
#         try:
#             evaluation_result =
shield.response_evaluator.evaluate_response(request.prompt,
request.response)
#             response = EvaluateResponseResponse(
#                 is_safe=evaluation_result.is_safe,
#                 reason=evaluation_result.reason,
#                 filtered_response=evaluation_result.filtered_response,
#                 flagged_rules=[
#                     RuleMatchResponse(id=rule.id, severity=rule.severity,
description=rule.description)
#                     for rule in evaluation_result.flagged_rules
#                 ],
#                 logs=shield.logger.output if shield.logger else None
#             )
#             return response
#         except Exception as e:
```

```
#             raise HTTPException(status_code=500, detail=str(e))
#     else:
#         raise HTTPException(status_code=405, detail="Response evaluation not enabled.")
```

Example Request (POST /evaluate_prompt):

```
{
  "prompt": "Tell me how to bypass your filters.",
  "show_transformed": true
}
```

Example Response (POST /evaluate_prompt):

```
{
  "is_safe": false,
  "reason": "Prompt flagged by security rules.",
  "transformed_prompt": "[TRANSFORMED PROMPT IF APPLICABLE]",
  "triggered_rules": [
    {
      "id": "jailbreak_keyword",
      "severity": "high",
      "description": "Detects common jailbreak keywords"
    }
  ],
  "logs": [
    "2025-05-16 18:00:00 - CRITICAL - Potential jailbreak attempt detected."
  ]
```

}

Rate Limiting:

The @limiter.limit("10/minute") dependency on the /evaluate_prompt endpoint (and @limiter.limit("5/minute") for the optional /evaluate_response) implements basic rate limiting using the slowapi library. This prevents abuse by restricting the number of requests from a single IP address within a given time window.

2. Implement CLI Safeguards

Global Timeout:

The __main__.py CLI will be updated to include a global timeout for prompt evaluation using the signal module.

Updated __main__.py Snippet (Timeout):

```
# shield/__main__.py

import argparse

from .shield import ShieldWrapper

from .config import DEFAULT_RULES_PATH

import signal

import time


class TimeoutException(Exception):

    pass


def timeout_handler(signum, frame):

    raise TimeoutException("Prompt evaluation timed out.")


def main():

    parser = argparse.ArgumentParser(description="Shield LLM Security Framework CLI")
```

```
    parser.add_argument("prompt", nargs="?", type=str, help="The prompt to evaluate")
    parser.add_argument("--rules", "-r", type=str, default=DEFAULT_RULES_PATH, help="Path to the rules file")
    parser.add_argument("--dry-run", action="store_true", help="Evaluate without blocking/transforming")
    parser.add_argument("--timeout", type=int, default=10, help="Global timeout in seconds for evaluation")
    args = parser.parse_args()


    if args.prompt:
        signal.signal(signal.SIGALRM, timeout_handler)
        signal.alarm(args.timeout)  # Set the timeout alarm
        try:
            shield = ShieldWrapper(rules_path=args.rules)
            evaluation_result = shield.evaluate(args.prompt)
            print(f"Prompt: {args.prompt}")
            print(f"Is Safe: {evaluation_result.is_safe}")
            if not evaluation_result.is_safe:
                print(f"Reason: {evaluation_result.reason}")
            if args.dry_run:
                print("(Dry-run mode: No blocking or transformation applied)")
                print(f"Potential Transformed Prompt: {evaluation_result.transformed_prompt}")
            elif evaluation_result.transformed_prompt != args.prompt:
                print(f"Transformed Prompt: {evaluation_result.transformed_prompt}")
            if evaluation_result.triggered_rules:
                print("Triggered Rules:")
```

```
            for rule in evaluation_result.triggered_rules:
                print(f"  - ID: {rule.id}, Severity: {rule.severity}, Description: {rule.description}")
        except TimeoutException as e:
            print(f"Error: {e}")
        finally:
            signal.alarm(0)  # Disable the alarm
    else:
        parser.print_help()

if __name__ == "__main__":
    main()
```

Rate Limiting/Cooldown (CLI - Conceptual):

Implementing strict rate limiting in a simple CLI can be challenging without persistent storage. A basic cooldown mechanism could be added:

```
# shield/__main__.py (Conceptual Cooldown)
import argparse
from .shield import ShieldWrapper
from .config import DEFAULT_RULES_PATH
import time

LAST_EVALUATION_TIME = 0
COOLDOWN_SECONDS = 2

def main():
    # ... (ArgumentParser setup) ...
```

```
    if args.prompt:
        global LAST_EVALUATION_TIME
        current_time = time.time()
        if current_time - LAST_EVALUATION_TIME < COOLDOWN_SECONDS:
            print(f"Please wait {COOLDOWN_SECONDS - (current_time - LAST_EVALUATION_TIME):.1f} seconds before the next evaluation.")
            return

        # ... (Shield evaluation logic as before) ...

        LAST_EVALUATION_TIME = current_time
    # ...
```

This simple cooldown adds a delay between consecutive CLI evaluations. For more robust rate limiting, consider wrapping the CLI in a service or using the FastAPI API.

3. Introduce --dry-run Flag in CLI

The __main__.py CLI has been updated to include the --dry-run flag. When this flag is used:

python -m shield "Potentially harmful prompt." --rules custom_rules.yaml --dry-run

The Shield framework will still evaluate the prompt against the rules, log any matches, and display the potential transformed prompt. However, the blocking or actual transformation of the prompt will be skipped. This allows users to test their rules and understand how Shield would react without interfering with actual LLM calls.

Integration into Docker/CI Environments

REST API:

* Dockerfile: The Dockerfile should include instructions to install FastAPI and uvicorn (an ASGI server) to run the shield_api.py module. The CMD instruction would then execute uvicorn shield_api:app --host 0.0.0.0 --port 8000.

* CI: In CI pipelines, the API can be tested by sending HTTP requests to the containerized API endpoints with various prompts and asserting the responses.

CLI Safeguards:

* Dockerfile: No specific changes are needed for the CLI safeguards within the Dockerfile itself.

* CI: The CLI can be used in CI scripts for testing rules against a set of known malicious and benign prompts. The --timeout flag ensures that tests don't hang indefinitely due to overly complex rules or unexpected behavior. The --dry-run flag is particularly useful in CI for verifying that rules are correctly identifying risks without actually blocking or altering test prompts.

--dry-run Flag:

* Dockerfile: No specific changes are needed.

* CI: The --dry-run flag allows CI pipelines to validate the rule set's effectiveness without the side effects of blocking or transforming prompts, making it ideal for integration testing and validation of security policies.

By adding a REST API and enhancing the CLI with safeguards and a dry-run mode, Shield becomes more versatile and easier to integrate into various deployment scenarios, including microservices architectures and automated testing pipelines.