Supercharge SAST

Semgrep Strategies for Secure Software

Agenda

- SAST and AppSec
- Getting started with Semgrep
- Writing custom rules
- Advanced Semgrep features
- Practical exercises
- Wrap-up
- Q&A

About Us

Software Security Engineering @ Microsoft

Arjun
https://www.linkedin.com/in/247arjun

Marcelo
https://www.linkedin.com/in/mribeirobr

Gautam
https://www.linkedin.com/in/perigautam

SAST and Application Security

MODULE 1

SAST and Application Security

Understanding SAST: Concepts and significance in securing software.

The role of SAST in the Software Development Life Cycle (SDLC).

What is SAST?

Static Application Security Testing (SAST) is a testing methodology that analyzes source code to find security vulnerabilities in applications

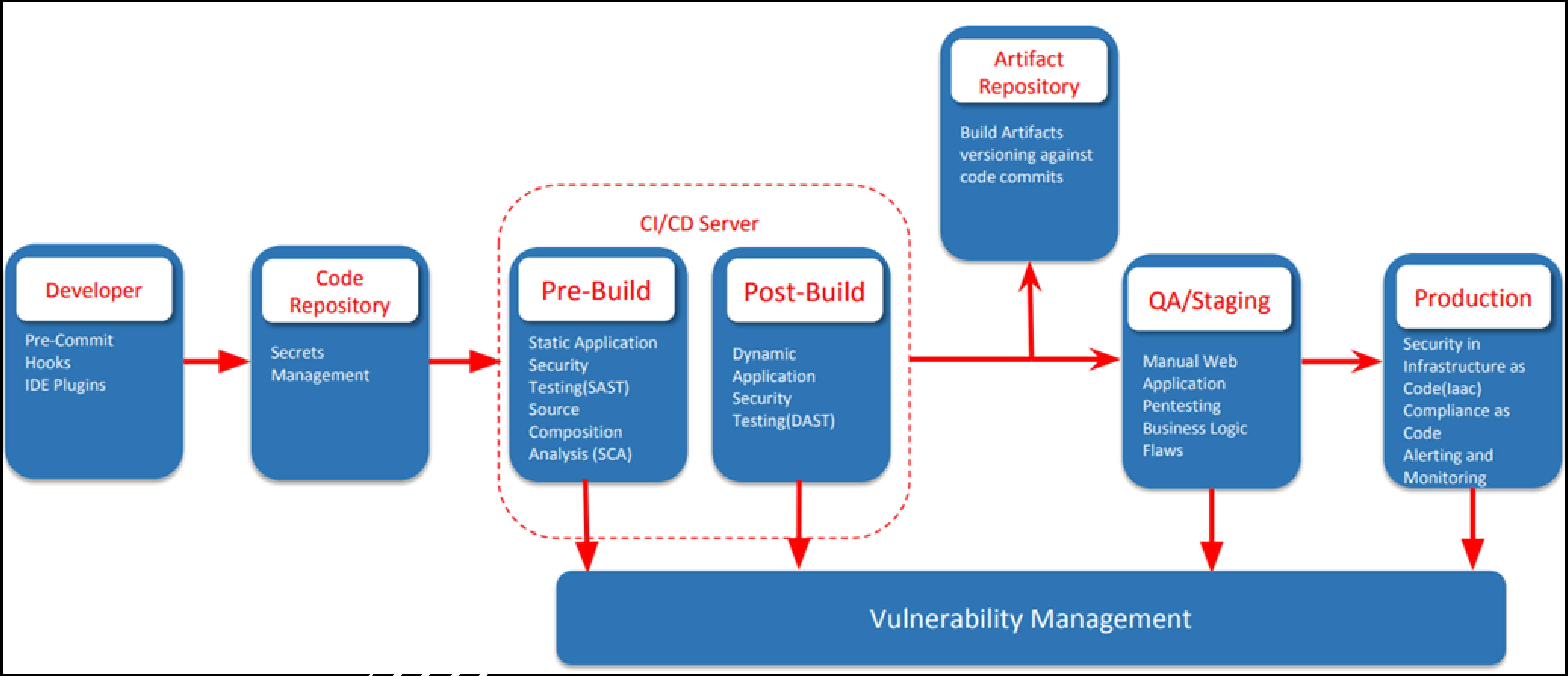
- Depends on patterns to define a vulnerability, also known as **signatures**

SAST can be used in real-time, as the developers code, it can be integrated into the DevOps cycle, or it can be executed offline.

As it is heavily dependent on signatures, it could lead to false positives if the signature base are not verified and tested to reduce de false alerts rate.

- LLMs can also be leveraged to reduce the false positive/negative rates

Typical SDLC


Artifact Repository
Build Artifacts versioning against code commits
CI/CD Server
Developer
Pre-Commit Hooks
IDE Plugins
Code Repository
Secrets Management
Pre-Build
Static Application Security Testing(SAST)
Source Composition Analysis (SCA)
Post-Build
Dynamic Application Security Testing(DAST)
QA/Staging
Manual Web Application Pentesting
Business Logic Flaws
Production
Security in Infrastructure as Code(Iaac)
Compliance as Code
Alerting and Monitoring
Vulnerability Management

Getting Started with Semgrep

MODULE 2

Getting Started with Semgrep

- Installing and configuring Semgrep.
- Navigating the Semgrep ecosystem: CLI and Playground.

Installing and getting ready

Install Semgrep on Linux, Windows and macOS

Pre-Requisite: Having Python 3.8 pre-installed

With all those OS, we can install Semgrep with PIP (make sure you have PIP installed)

```
python3 –m pip install semgrep
```

Installing Semgrep using other methods (Homebrew on macOS) or use the Semgrep Docker container to run it are options, but those methods won’t be covered on this workshop.

Semgrep… Installing and getting ready

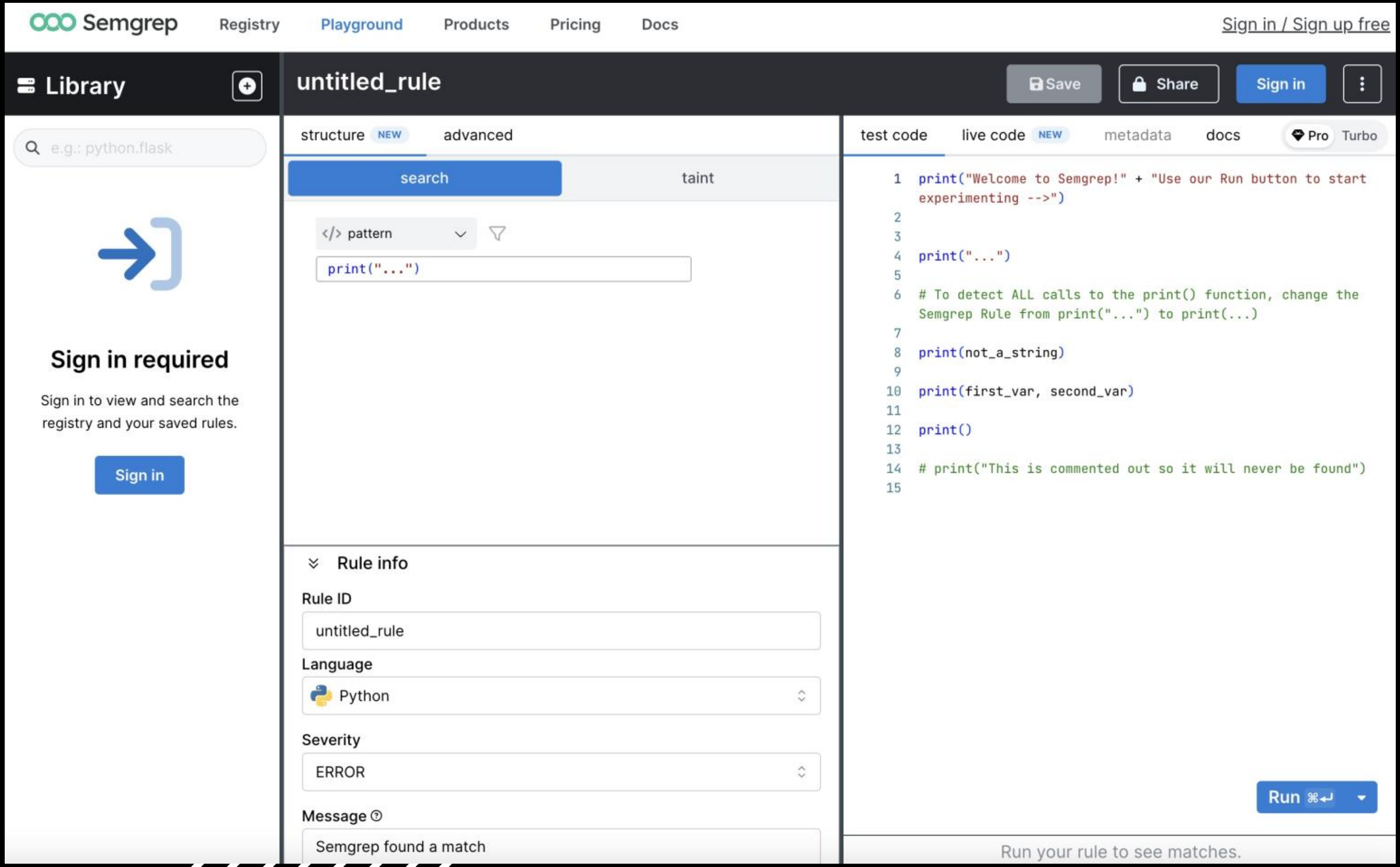
- Two use modes:
  - Online rules with Semgrep web interface (semgrep.dev)
    - The GUI helps with analysis over the findings
    - It has a free tier, but not for the PRO ruleset
  - Local rules
    - You can use the default ruleset, and add local rules set you built or downloaded
    - It generates text-formatted outputs with the findings
    - It does not provide a GUI to help with analysis… (but you can always feed the results into an ELK/Spluk for analysis…)

- We will focus on features available on the Open Source ONLY!

Navigating the Semgrep ecosystem: CLI and Playground

- Basic CLI Command:
  - semgrep scan [options] [targets]
- Some interesting Semgrep CLI options:
  - --config auto – Downloads the relevant rules for the current project from Semgrep opensource rules repository
  - --config <path_to_rules> - Points to local custom rulesets
  - Output formats
    - --json – Output to JSON formatted file
    - --text – Output to a text formatted file
    - --vim – Output to a vim-friendly formatted file
    - --sarif – Output to a sarif formatted file

Semgrep Playground (https://semgrep.dev/playground/new)



Sign in required

Sign in to view and search the registry and your saved rules.

Sign in

Pattern:

```
print("...")
```

Rule ID: untitled_rule

Language: Python

Severity: ERROR

Message: Semgrep found a match

```python
print("Welcome to Semgrep!" + "Use our Run button to start experimenting -->")


print("...")

# To detect ALL calls to the print() function, change the Semgrep Rule from print("...") to print(...)

print(not_a_string)

print(first_var, second_var)

print()

# print("This is commented out so it will never be found")

```

Run your rule to see matches.

Writing Custom Semgrep Rules

MODULE 3

Writing Custom Semgrep Rules

Deep dive into Semgrep's rule syntax and pattern matching.

Semgrep Rules

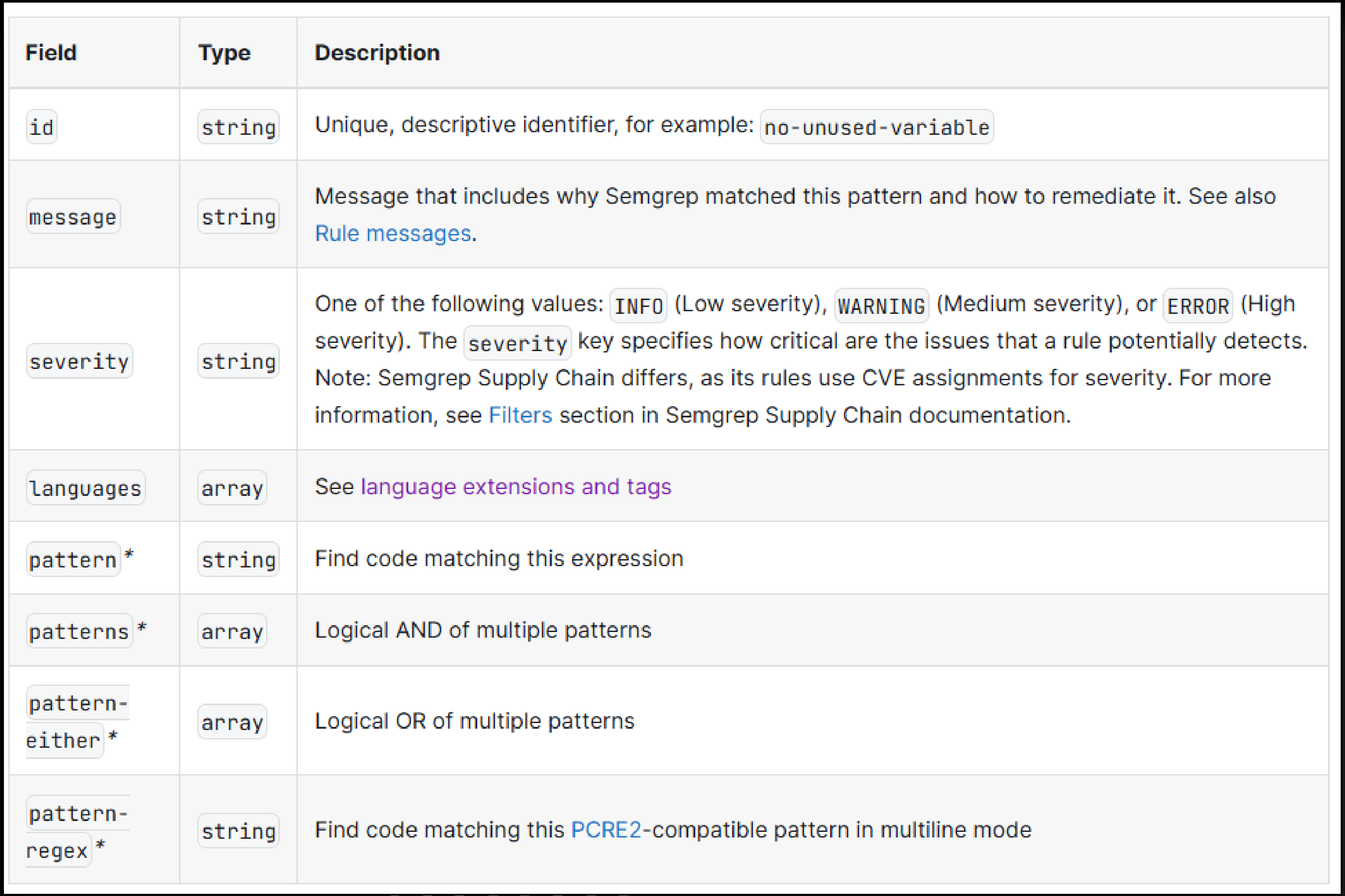
- *A <u>rule is a specification of the patterns</u>* that Semgrep must match to the code to generate a finding.

- Semgrep Rules are written in YAML

- Semgrep supports multiple languages
  - e.g.: Python, Java, C#, JavaScript, Go, etc..

- Multiple modes
  - Search (default)
  - Taint
  - Join
  - Extract

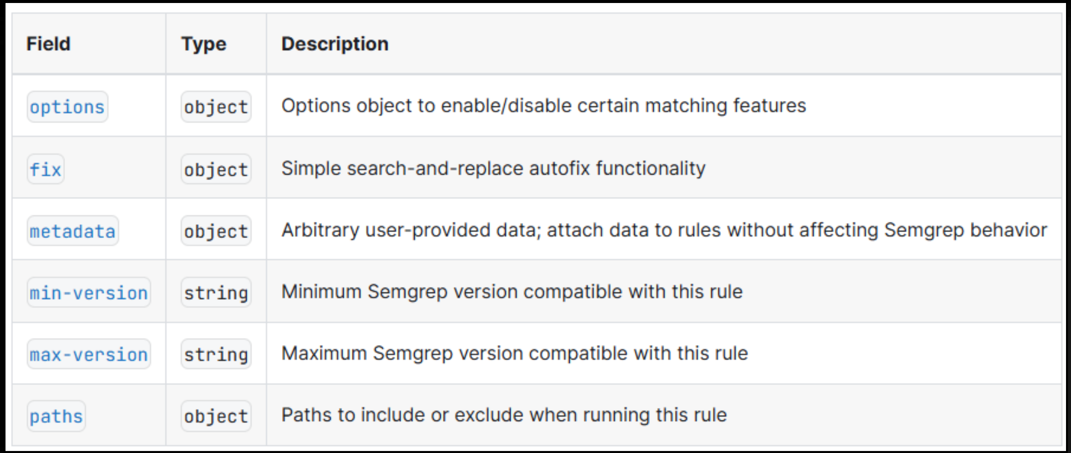
Schema – Required Fields



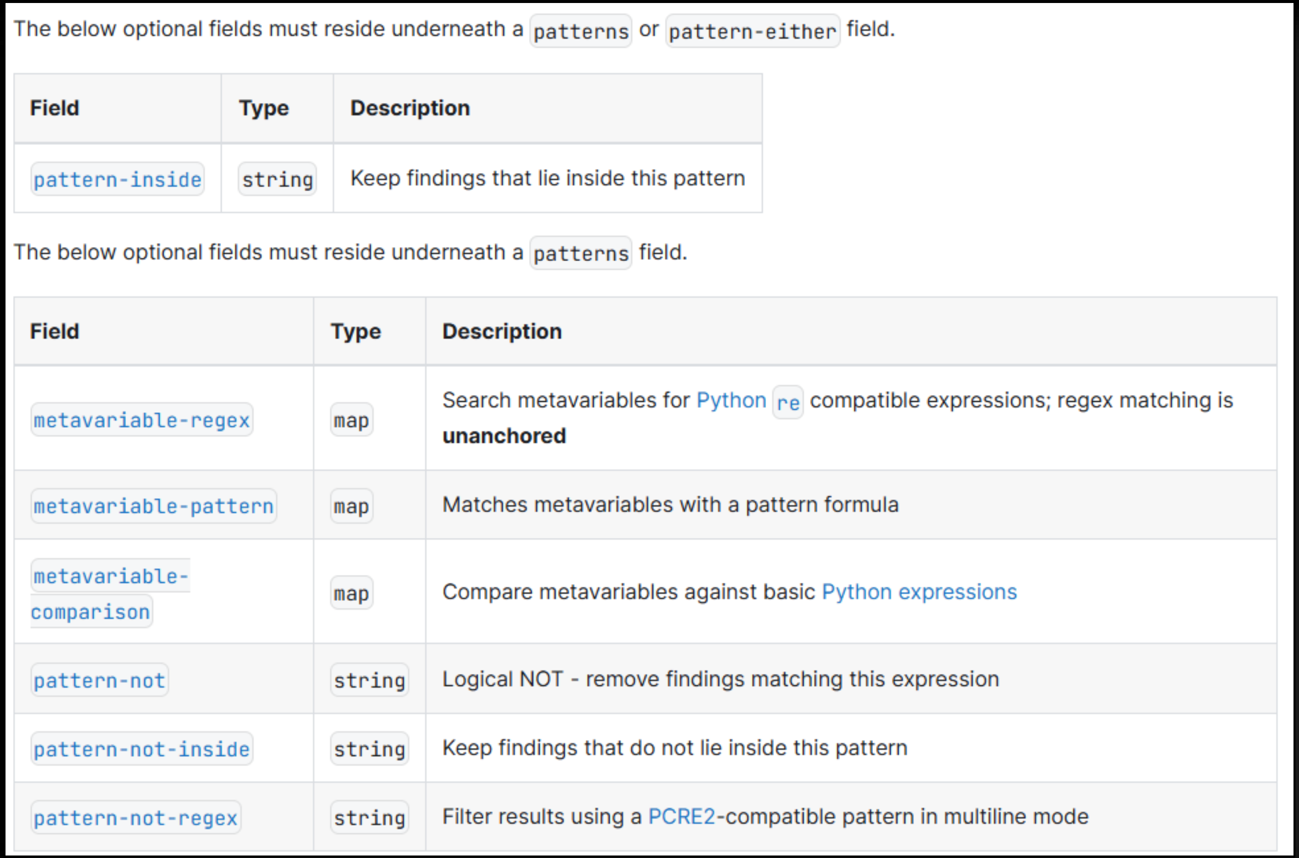
Field	Type	Description
`id`	`string`	Unique, descriptive identifier, for example: `no-unused-variable`
`message`	`string`	Message that includes why Semgrep matched this pattern and how to remediate it. See also Rule messages.
`severity`	`string`	One of the following values: `INFO` (Low severity), `WARNING` (Medium severity), or `ERROR` (High severity). The `severity` key specifies how critical are the issues that a rule potentially detects. Note: Semgrep Supply Chain differs, as its rules use CVE assignments for severity. For more information, see Filters section in Semgrep Supply Chain documentation.
`languages`	`array`	See language extensions and tags
`pattern` *	`string`	Find code matching this expression
`patterns` *	`array`	Logical AND of multiple patterns
`pattern-either` *	`array`	Logical OR of multiple patterns
`pattern-regex` *	`string`	Find code matching this PCRE2-compatible pattern in multiline mode

Rule syntax | Semgrep

Schema – Optional Fields



Field	Type	Description
`options`	`object`	Options object to enable/disable certain matching features
`fix`	`object`	Simple search-and-replace autofix functionality
`metadata`	`object`	Arbitrary user-provided data; attach data to rules without affecting Semgrep behavior
`min-version`	`string`	Minimum Semgrep version compatible with this rule
`max-version`	`string`	Maximum Semgrep version compatible with this rule
`paths`	`object`	Paths to include or exclude when running this rule



The below optional fields must reside underneath a `patterns` or `pattern-either` field.

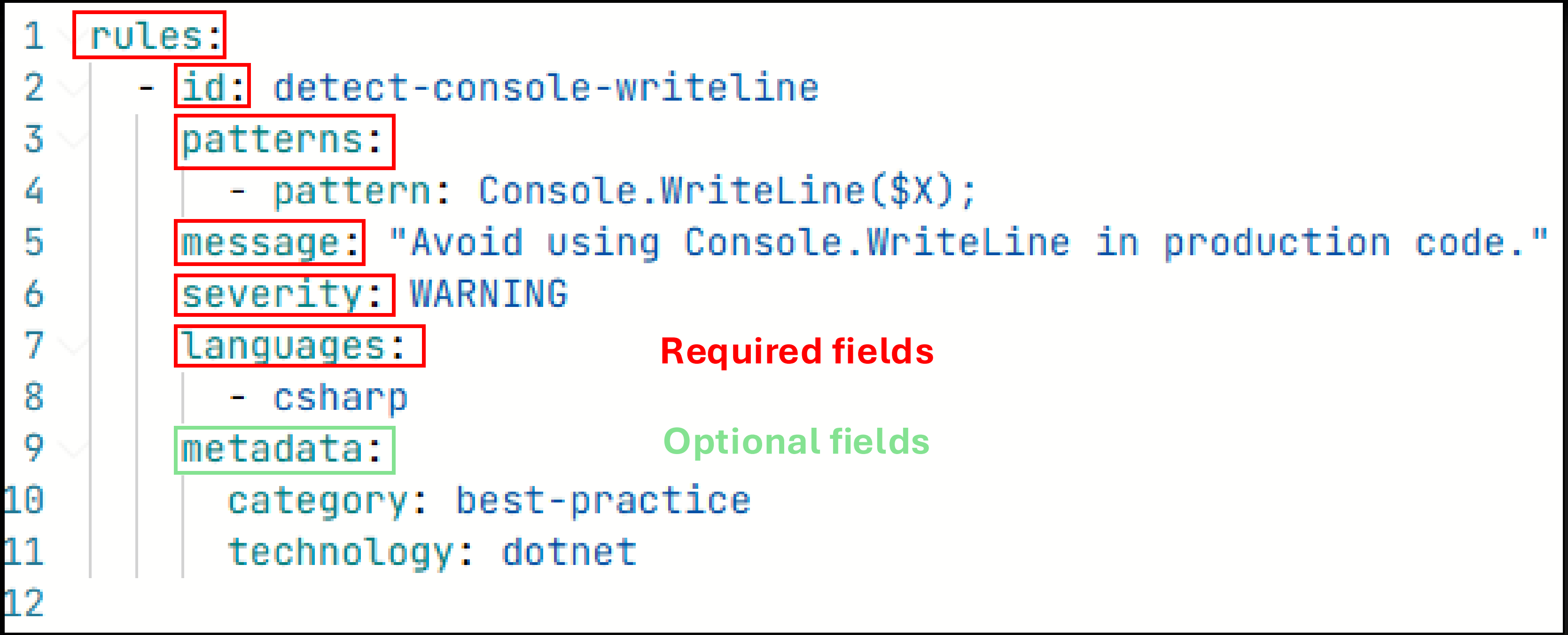
Field	Type	Description
`pattern-inside`	`string`	Keep findings that lie inside this pattern

The below optional fields must reside underneath a `patterns` field.

Field	Type	Description
`metavariable-regex`	`map`	Search metavariables for Python `re` compatible expressions; regex matching is **unanchored**
`metavariable-pattern`	`map`	Matches metavariables with a pattern formula
`metavariable-comparison`	`map`	Compare metavariables against basic Python expressions
`pattern-not`	`string`	Logical NOT - remove findings matching this expression
`pattern-not-inside`	`string`	Keep findings that do not lie inside this pattern
`pattern-not-regex`	`string`	Filter results using a PCRE2-compatible pattern in multiline mode

Rule syntax | Semgrep

Semgrep Rule Syntax



```yaml
rules:
  - id: detect-console-writeline
    patterns:
      - pattern: Console.WriteLine($X);
    message: "Avoid using Console.WriteLine in production code."
    severity: WARNING
    languages:
      - csharp
    metadata:
      category: best-practice
      technology: dotnet
```

Rule syntax | Semgrep

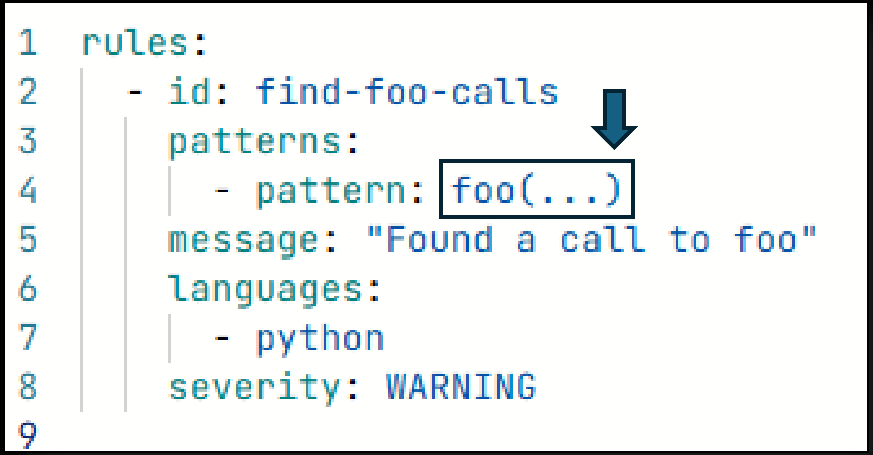
Pattern Syntax

- Ellipsis Operator (...)
- Metavariables ($VARNAME)

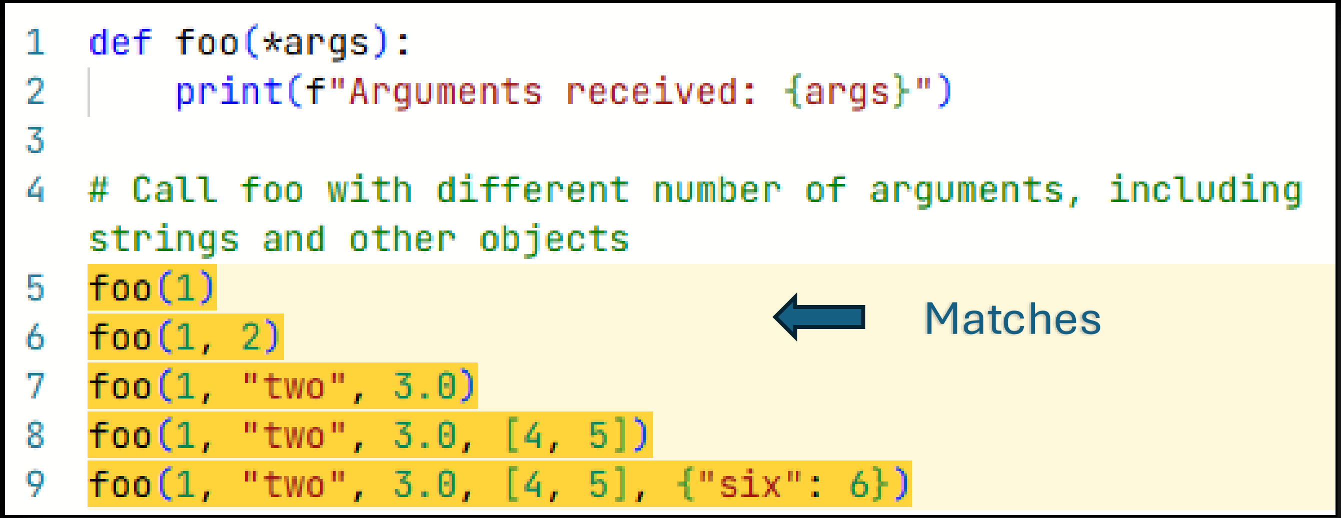
Rule syntax | Semgrep

Ellipsis Operator ...

- *Abstracts away* a sequence of zero or more items such as arguments, statements, parameters, fields, characters.



```yaml
rules:
  - id: find-foo-calls
    patterns:
      - pattern: foo(...)
    message: "Found a call to foo"
    languages:
      - python
    severity: WARNING
```



```python
def foo(*args):
    print(f"Arguments received: {args}")

# Call foo with different number of arguments, including strings and other objects
foo(1)
foo(1, 2)
foo(1, "two", 3.0)
foo(1, "two", 3.0, [4, 5])
foo(1, "two", 3.0, [4, 5], {"six": 6})
```

Rule syntax | Semgrep

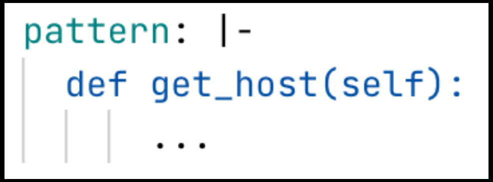
Ellipsis Operator ...

- Exercise - (aka.ms/semgrep-exercises)
  - https://github.com/django/django/blob/main/django/http/request.py
  - Flag and return the entire function body for
    - *def get_host(self):*
    - *def _get_full_path( ): – not knowing the function's arguments*

Rule syntax | Semgrep

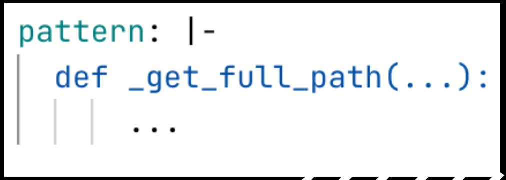
Ellipsis Operator ...

- Answers
  - Flag and return the entire function body for
    - *def get_host(self):*



```
pattern: |-
  def get_host(self):
    ...
```

    - *def _get_full_path – not knowing the function's arguments*

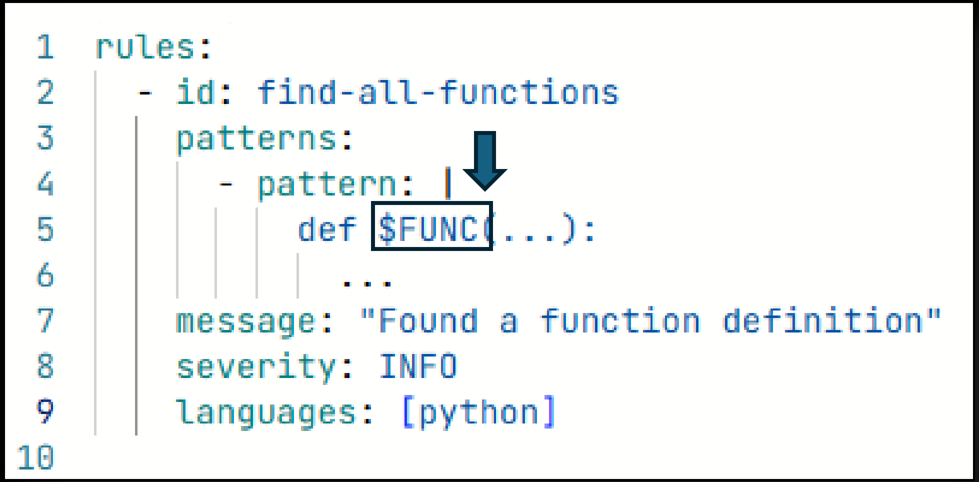


```
pattern: |-
  def _get_full_path(...):
    ...
```

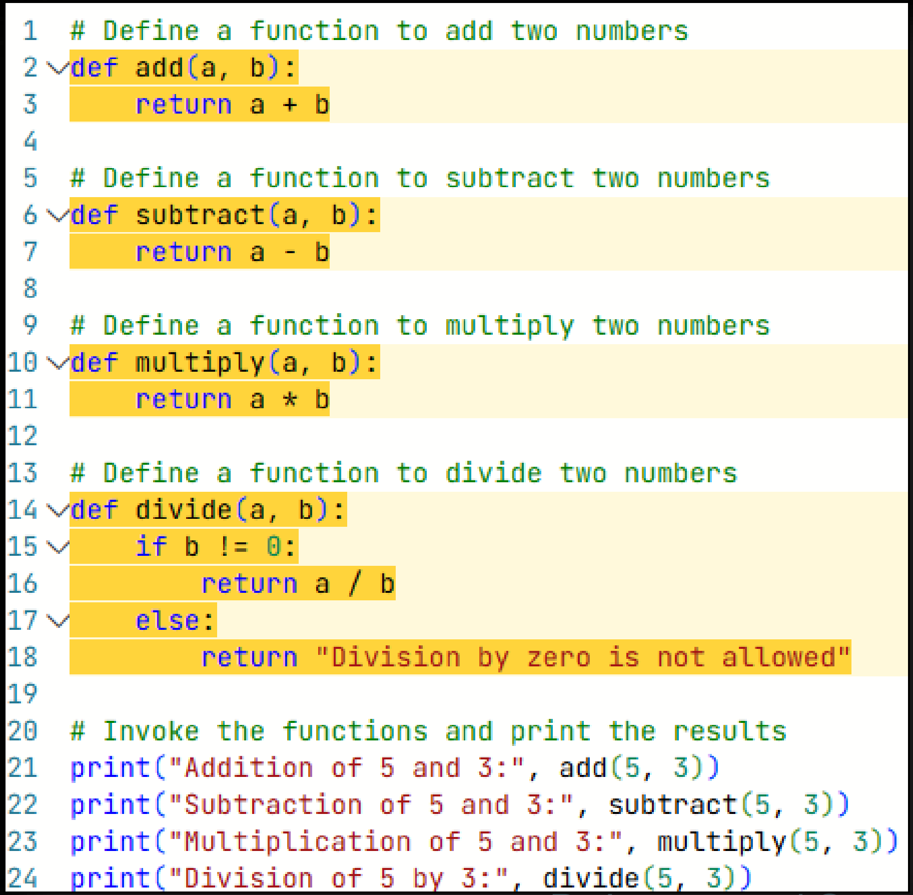
Rule syntax | Semgrep

Metavariables - $VARIABLE_NAME

- *Metavariables are an abstraction to match code* when you don't know the value or contents ahead of time



```yaml
rules:
  - id: find-all-functions
    patterns:
      - pattern: |
          def $FUNC(...):
            ...
    message: "Found a function definition"
    severity: INFO
    languages: [python]
```



```python
# Define a function to add two numbers
def add(a, b):
    return a + b

# Define a function to subtract two numbers
def subtract(a, b):
    return a - b

# Define a function to multiply two numbers
def multiply(a, b):
    return a * b

# Define a function to divide two numbers
def divide(a, b):
    if b != 0:
        return a / b
    else:
        return "Division by zero is not allowed"

# Invoke the functions and print the results
print("Addition of 5 and 3:", add(5, 3))
print("Subtraction of 5 and 3:", subtract(5, 3))
print("Multiplication of 5 and 3:", multiply(5, 3))
print("Division of 5 by 3:", divide(5, 3))
```

Rule syntax | Semgrep

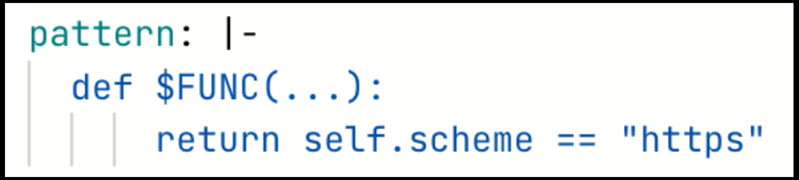
Metavariables - $VARIABLE_NAME

- Exercise - (aka.ms/semgrep-exercises)
  - https://github.com/django/django/blob/main/django/http/request.py
  - Flag and return the entire function bodies that contain the line
    - *return self.scheme == “https”*
    - *parser = MultiPartParser(…)*

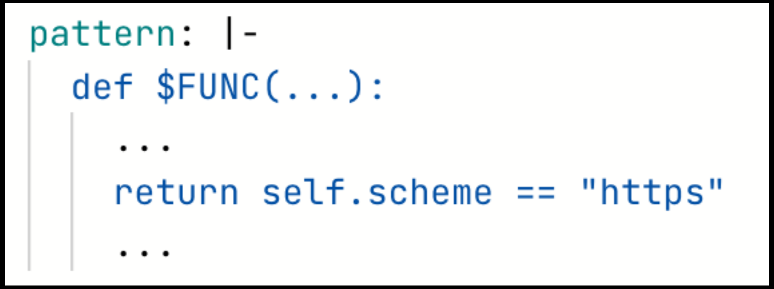
Rule syntax | Semgrep

Metavariables - $VARIABLE_NAME

- Answers
  - Flag and return the entire function bodies that contain the line
    - *return self.scheme == “https”*

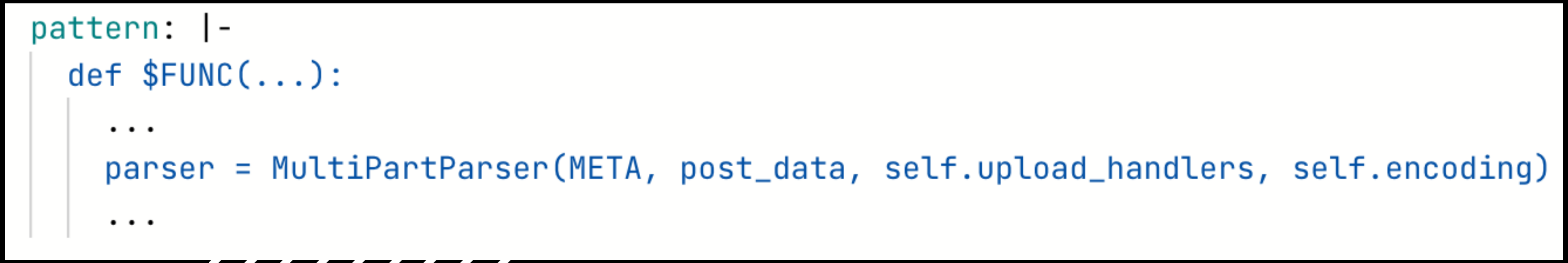


```
pattern: |-
  def $FUNC(...):
      return self.scheme == "https"
```



```
pattern: |-
  def $FUNC(...):
    ...
    return self.scheme == "https"
    ...
```

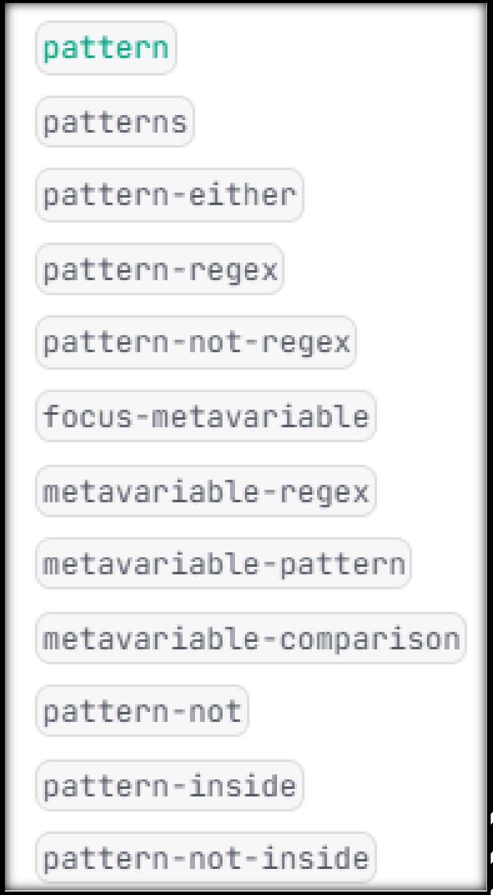
    - *parser = MultiPartParser(META, post_data, self.upload_handlers, self.encoding)*

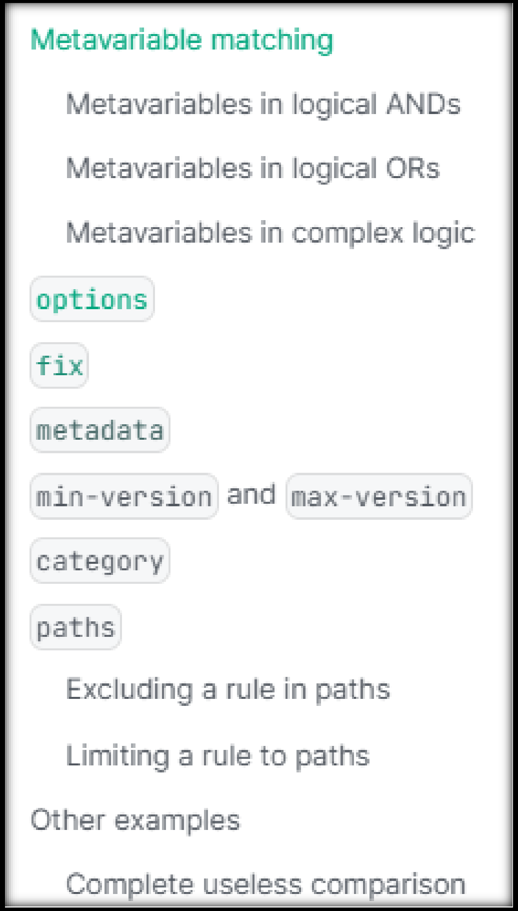


```
pattern: |-
  def $FUNC(...):
    ...
    parser = MultiPartParser(META, post_data, self.upload_handlers, self.encoding)
    ...
```

Rule syntax | Semgrep

Pattern Matching Operators

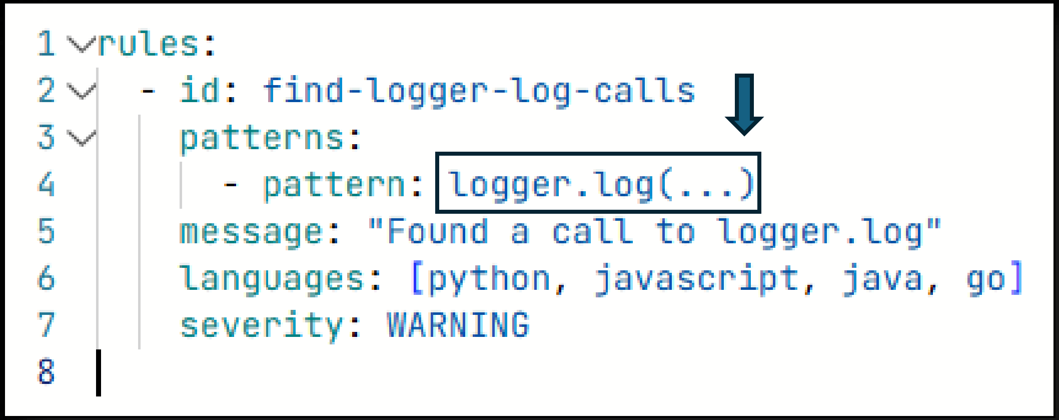




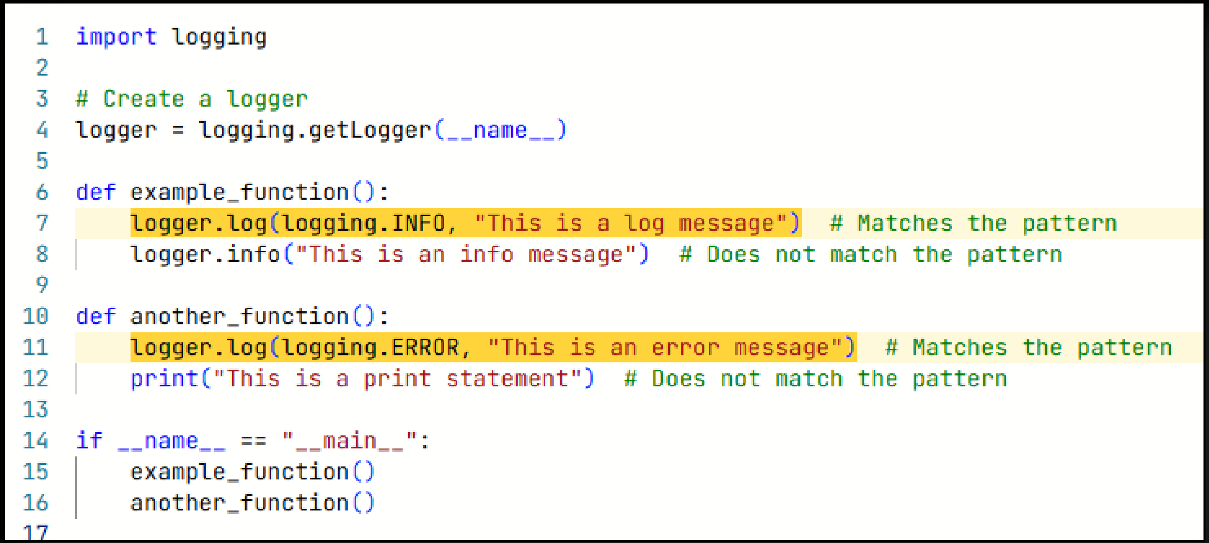
[Rule syntax | Semgrep]

Pattern

- The *pattern* operator looks for code matching its expression.



```yaml
rules:
  - id: find-logger-log-calls
    patterns:
      - pattern: logger.log(...)
    message: "Found a call to logger.log"
    languages: [python, javascript, java, go]
    severity: WARNING

```

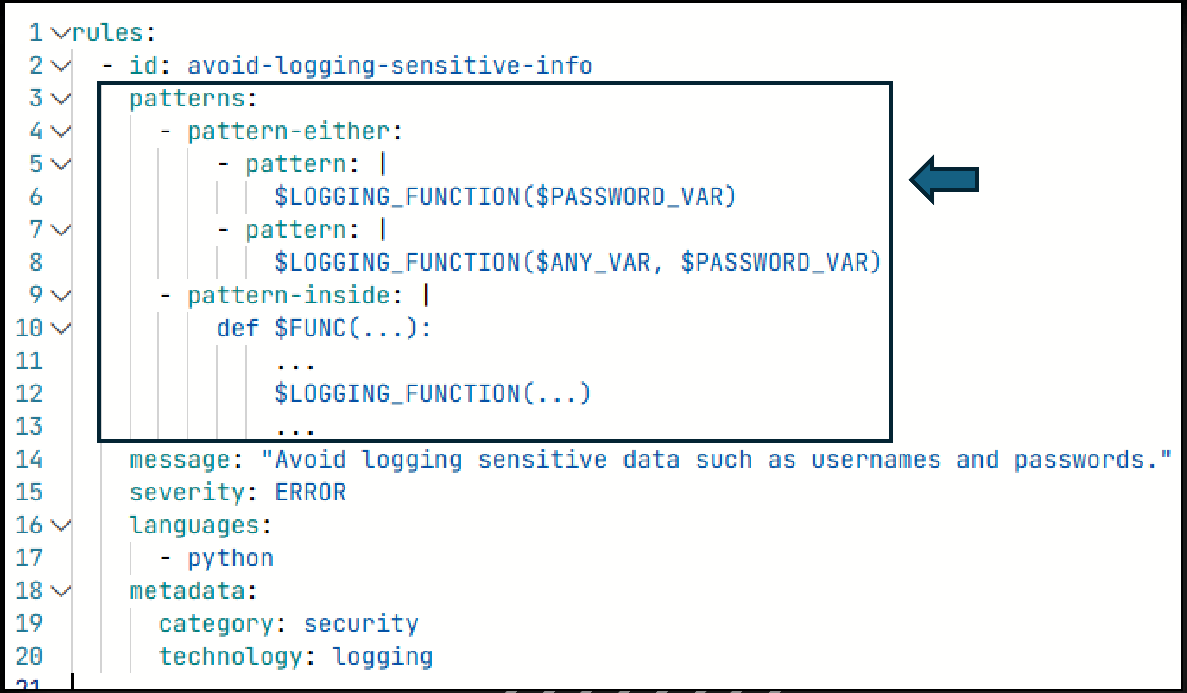


```python
import logging

# Create a logger
logger = logging.getLogger(__name__)

def example_function():
    logger.log(logging.INFO, "This is a log message")  # Matches the pattern
    logger.info("This is an info message")  # Does not match the pattern

def another_function():
    logger.log(logging.ERROR, "This is an error message")  # Matches the pattern
    print("This is a print statement")  # Does not match the pattern

if __name__ == "__main__":
    example_function()
    another_function()
```

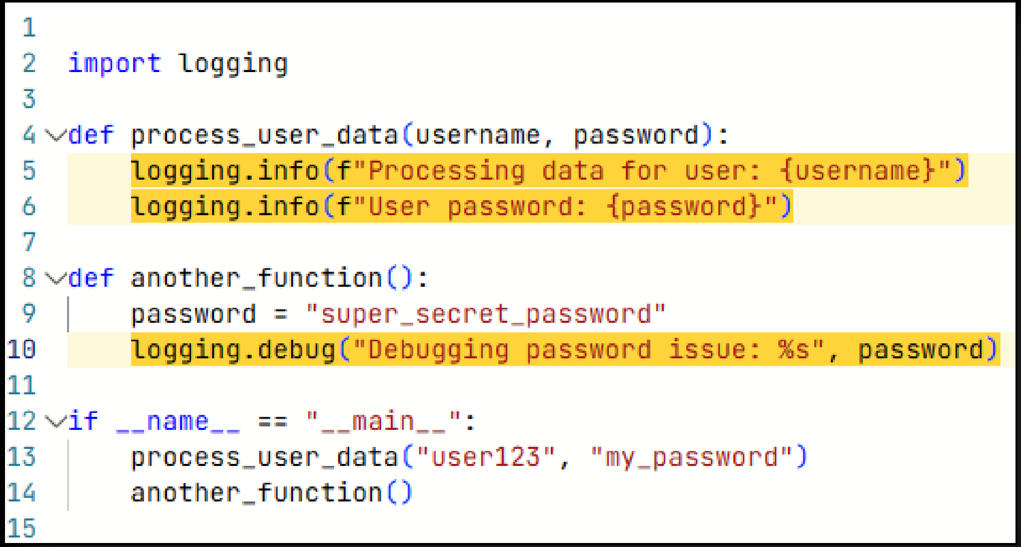
Rule syntax | Semgrep

Patterns

- Performs a logical AND operation on one or more child patterns.
- Chaining multiple patterns together



```yaml
rules:
  - id: avoid-logging-sensitive-info
    patterns:
      - pattern-either:
          - pattern: |
              $LOGGING_FUNCTION($PASSWORD_VAR)
          - pattern: |
              $LOGGING_FUNCTION($ANY_VAR, $PASSWORD_VAR)
      - pattern-inside: |
          def $FUNC(...):
              ...
              $LOGGING_FUNCTION(...)
              ...
    message: "Avoid logging sensitive data such as usernames and passwords."
    severity: ERROR
    languages:
      - python
    metadata:
      category: security
      technology: logging
```



```python

import logging

def process_user_data(username, password):
    logging.info(f"Processing data for user: {username}")
    logging.info(f"User password: {password}")

def another_function():
    password = "super_secret_password"
    logging.debug("Debugging password issue: %s", password)

if __name__ == "__main__":
    process_user_data("user123", "my_password")
    another_function()

```

Rule syntax | Semgrep

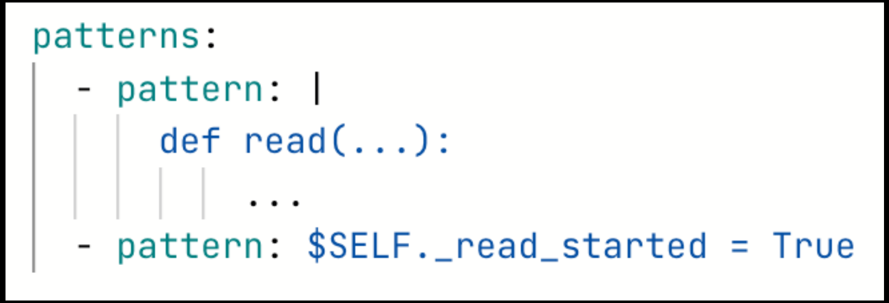
Patterns

- Exercise - (aka.ms/semgrep-exercises)
  - https://github.com/django/django/blob/main/django/http/request.py
  - Using 1 rule, flag and return the line of code
    - That is inside a function named *read*
    - And sets the *_read_started* attribute to *True*

Rule syntax | Semgrep

Patterns

- Answer
  - Using 1 rule, flag and return the line of code
    - That is inside a function named *read*
    - And sets the *_read_started* attribute to *True*

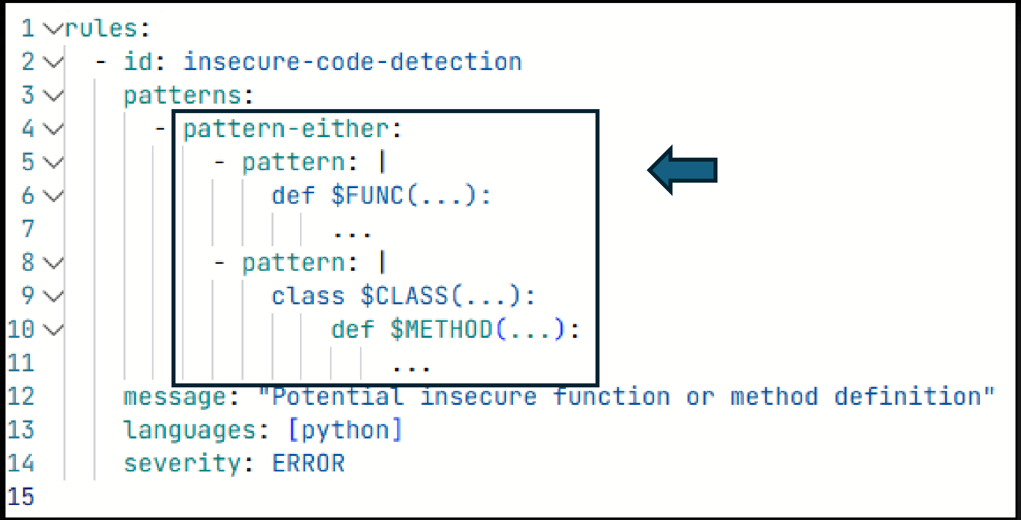


```
patterns:
  - pattern: |
      def read(...):
          ...
  - pattern: $SELF._read_started = True
```

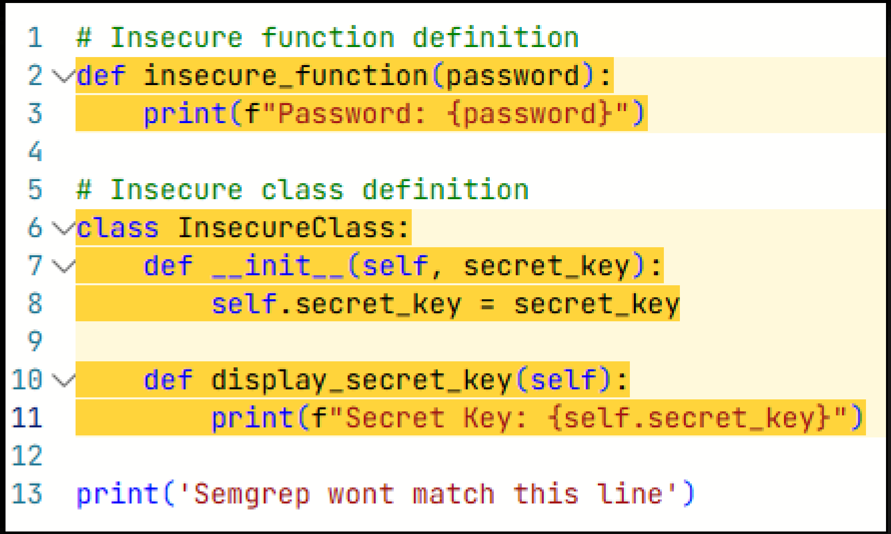
Rule syntax | Semgrep

Pattern-Either

- Performs a logical OR operation on one or more child patterns.
- Chaining multiple patterns together



```yaml
rules:
  - id: insecure-code-detection
    patterns:
      - pattern-either:
        - pattern: |
            def $FUNC(...):
              ...
        - pattern: |
            class $CLASS(...):
              def $METHOD(...):
                ...
    message: "Potential insecure function or method definition"
    languages: [python]
    severity: ERROR
```



```python
# Insecure function definition
def insecure_function(password):
    print(f"Password: {password}")

# Insecure class definition
class InsecureClass:
    def __init__(self, secret_key):
        self.secret_key = secret_key

    def display_secret_key(self):
        print(f"Secret Key: {self.secret_key}")

print('Semgrep wont match this line')
```

Rule syntax | Semgrep

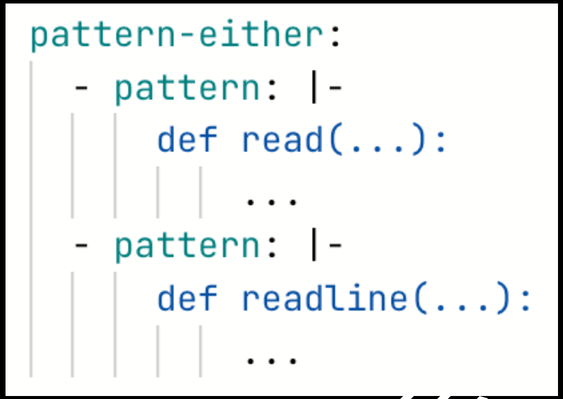
Pattern-Either

- Exercise - (aka.ms/semgrep-exercises)
  - https://github.com/django/django/blob/main/django/http/request.py
  - Using 1 rule, flag and return both functions
    - A function named *read*
    - A function named *readline*

Rule syntax | Semgrep

Pattern-Either

- Answer
  - Using 1 rule, flag and return both functions
    - A function named *read*
    - A function named *readline*

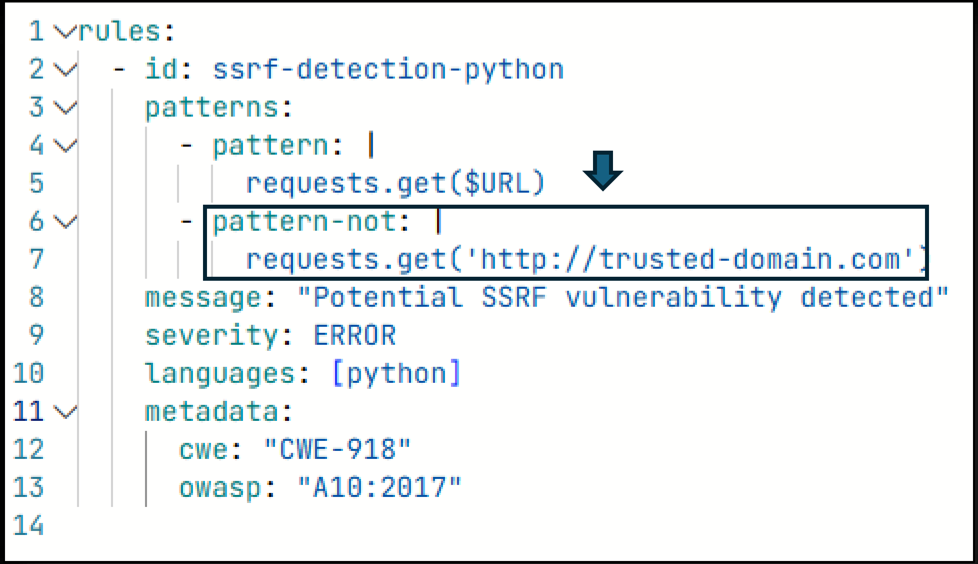


```
pattern-either:
  - pattern: |-
      def read(...):
          ...
  - pattern: |-
      def readline(...):
          ...
```

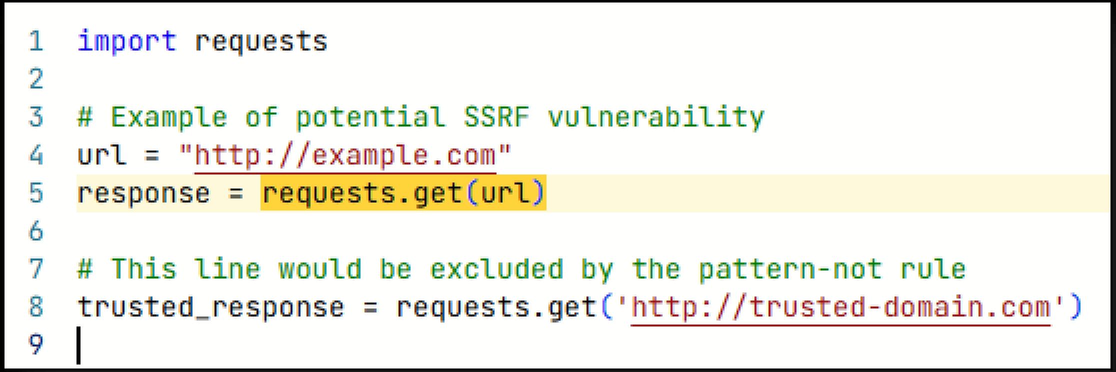
Rule syntax | Semgrep

Pattern-Not

- The *pattern-not* operator is the opposite of the pattern operator.
- Finds code that does **NOT** match its expression



```yaml
rules:
  - id: ssrf-detection-python
    patterns:
      - pattern: |
          requests.get($URL)
      - pattern-not: |
          requests.get('http://trusted-domain.com')
    message: "Potential SSRF vulnerability detected"
    severity: ERROR
    languages: [python]
    metadata:
      cwe: "CWE-918"
      owasp: "A10:2017"
```



```python
import requests

# Example of potential SSRF vulnerability
url = "http://example.com"
response = requests.get(url)

# This line would be excluded by the pattern-not rule
trusted_response = requests.get('http://trusted-domain.com')
```

Rule syntax | Semgrep

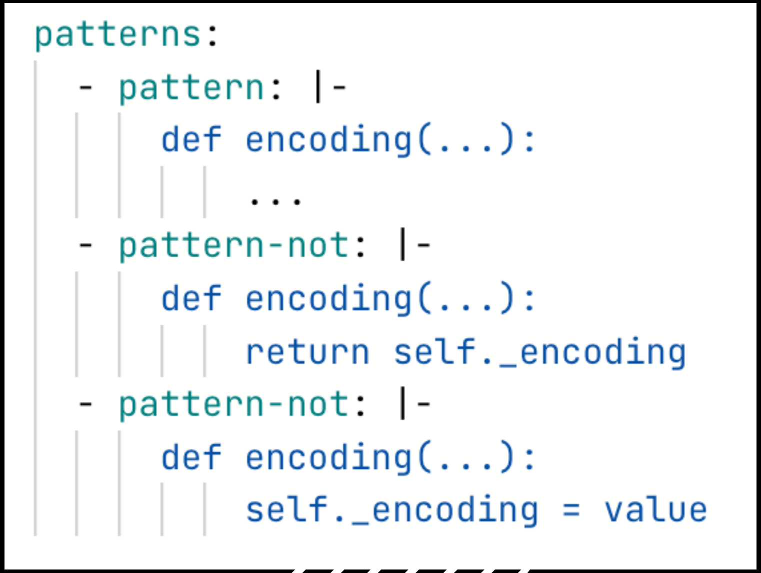
Pattern-Not

- Exercise - (aka.ms/semgrep-exercises)
  - https://github.com/django/django/blob/main/django/http/request.py
  - Using 1 rule, flag and return both functions
    - All functions named *encoding*
    - But not those that are one-liners
      - *return self._encoding*
      - *self._encoding = value*

Rule syntax | Semgrep

Pattern-Not

- Answer
  - Using 1 rule, flag and return both functions
    - All functions named *encoding*
    - But not those that are one-liners
      - *return self._encoding*
      - *self._encoding = value*

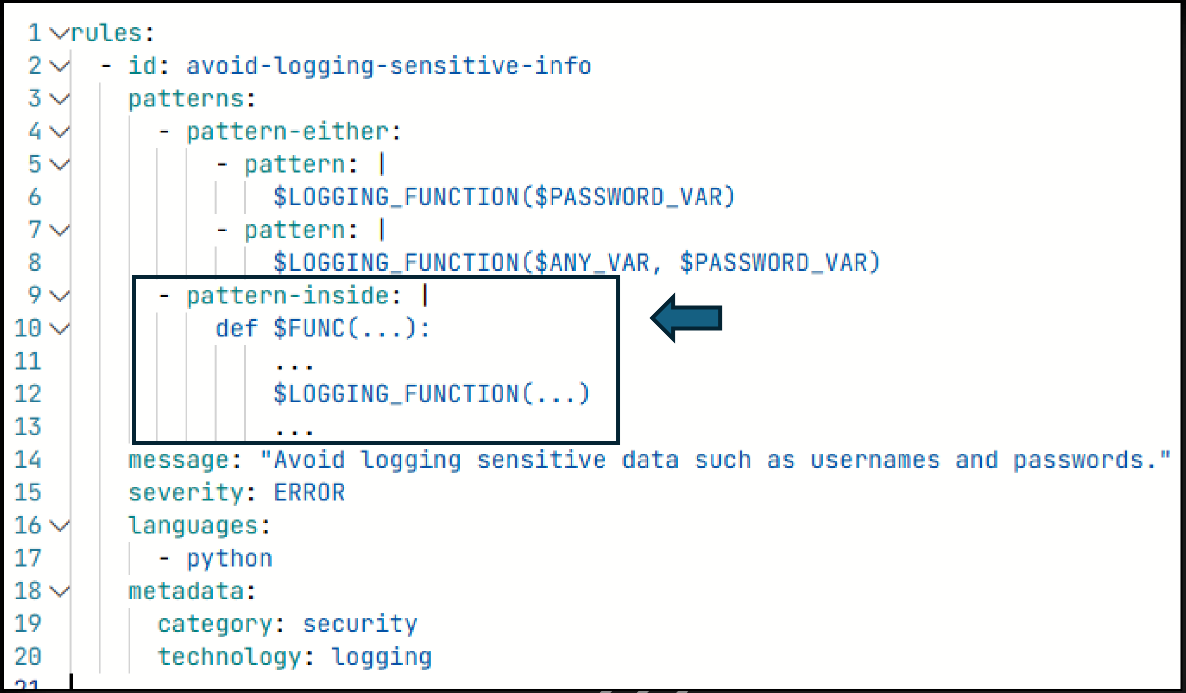


```yaml
patterns:
  - pattern: |-
      def encoding(...):
          ...
  - pattern-not: |-
      def encoding(...):
          return self._encoding
  - pattern-not: |-
      def encoding(...):
          self._encoding = value
```

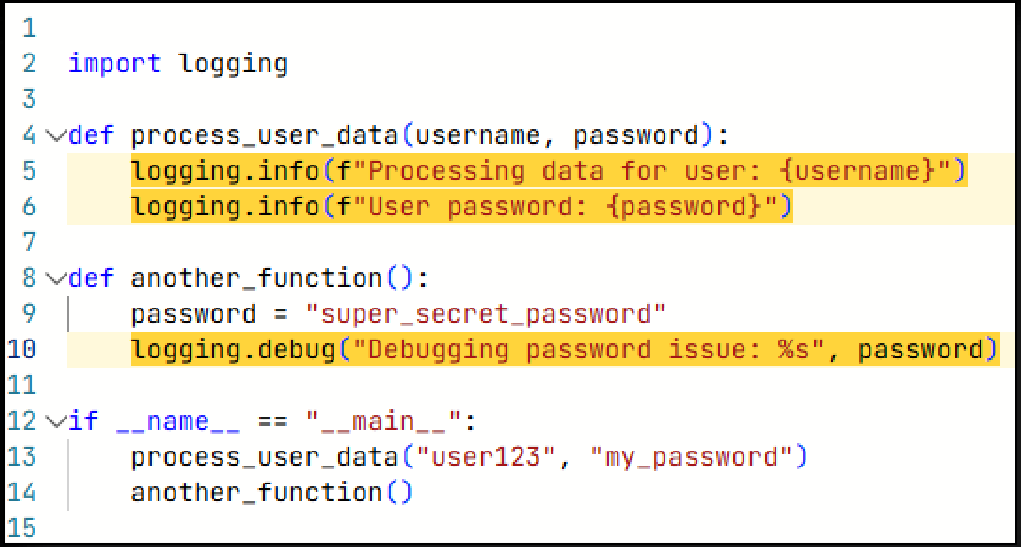
Rule syntax | Semgrep

Pattern-Inside

- Matches findings that reside within its expression



```yaml
rules:
  - id: avoid-logging-sensitive-info
    patterns:
      - pattern-either:
          - pattern: |
              $LOGGING_FUNCTION($PASSWORD_VAR)
          - pattern: |
              $LOGGING_FUNCTION($ANY_VAR, $PASSWORD_VAR)
      - pattern-inside: |
          def $FUNC(...):
            ...
            $LOGGING_FUNCTION(...)
            ...
    message: "Avoid logging sensitive data such as usernames and passwords."
    severity: ERROR
    languages:
      - python
    metadata:
      category: security
      technology: logging
```



```python

import logging

def process_user_data(username, password):
    logging.info(f"Processing data for user: {username}")
    logging.info(f"User password: {password}")

def another_function():
    password = "super_secret_password"
    logging.debug("Debugging password issue: %s", password)

if __name__ == "__main__":
    process_user_data("user123", "my_password")
    another_function()

```

[Rule syntax | Semgrep]

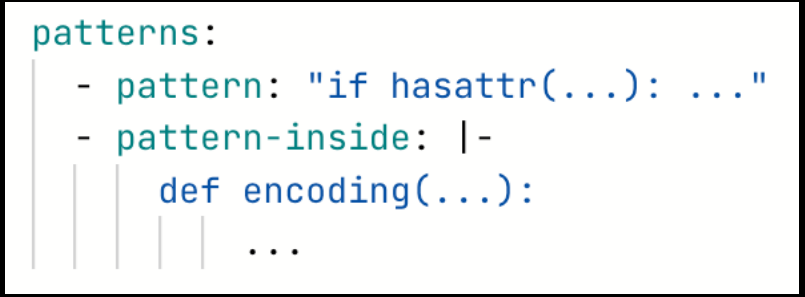
Pattern-Inside

- Exercise - (aka.ms/semgrep-exercises)
  - https://github.com/django/django/blob/main/django/http/request.py
  - Using 1 rule, flag and return
    - All instances of *if hasattr(...): ...*
    - That are inside a function named *encoding*

Rule syntax | Semgrep

Pattern-Inside

- Answer
  - Using 1 rule, flag and return
    - All instances of *if hasattr(...): ...*
    - That are inside a function named *encoding*

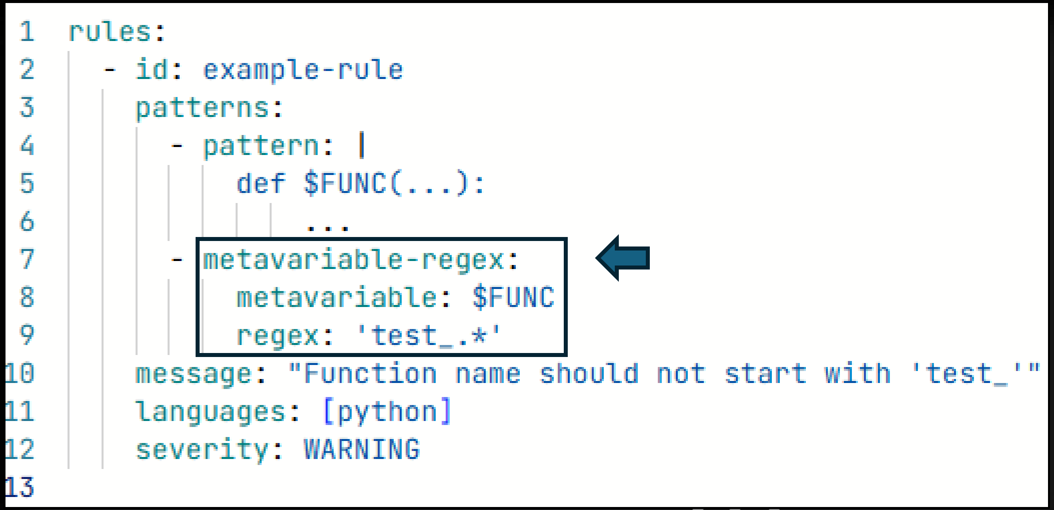


```
patterns:
  - pattern: "if hasattr(...): ..."
  - pattern-inside: |-
      def encoding(...):
          ...
```

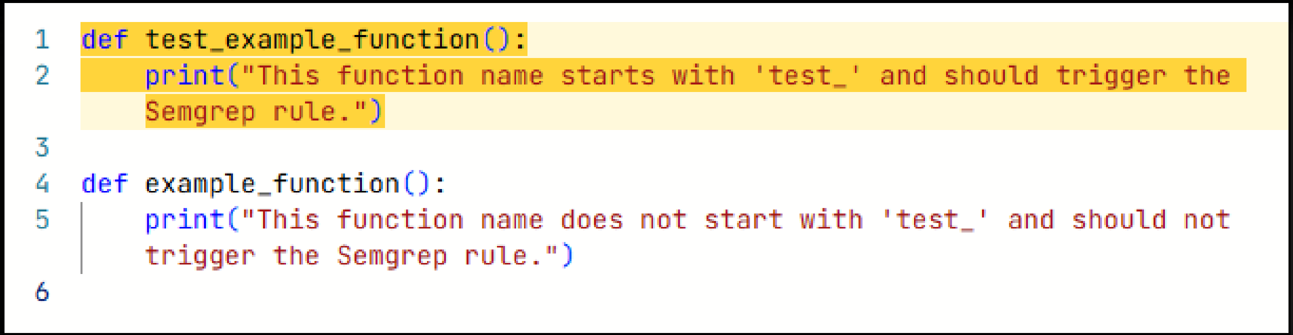
Rule syntax | Semgrep

Metavariable regex

- *Searches metavariables for a PCRE2 regular expression*
- *Useful in filtering results based on a metavariable's value*



```yaml
rules:
  - id: example-rule
    patterns:
      - pattern: |
          def $FUNC(...):
            ...
      - metavariable-regex:
          metavariable: $FUNC
          regex: 'test_.*'
    message: "Function name should not start with 'test_'"
    languages: [python]
    severity: WARNING
```



```python
def test_example_function():
    print("This function name starts with 'test_' and should trigger the Semgrep rule.")

def example_function():
    print("This function name does not start with 'test_' and should not trigger the Semgrep rule.")
```

Rule syntax | Semgrep

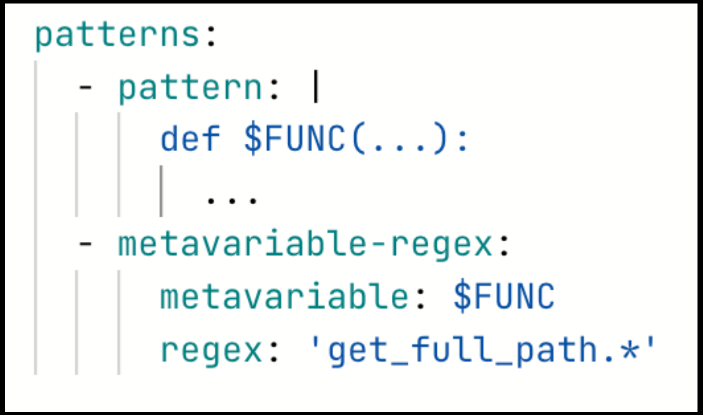
Metavariable-regex

- Exercise - (aka.ms/semgrep-exercises)
  - https://github.com/django/django/blob/main/django/http/request.py
  - Using 1 rule, flag and return
    - All function names that begin with *get_full_path*

Rule syntax | Semgrep

Metavariable-regex

- Answer
  - Using 1 rule, flag and return all function names that begin with *get_full_path*



```
patterns:
  - pattern: |
      def $FUNC(...):
        ...
  - metavariable-regex:
      metavariable: $FUNC
      regex: 'get_full_path.*'
```

Rule syntax | Semgrep

Advanced Semgrep Features

MODULE 4

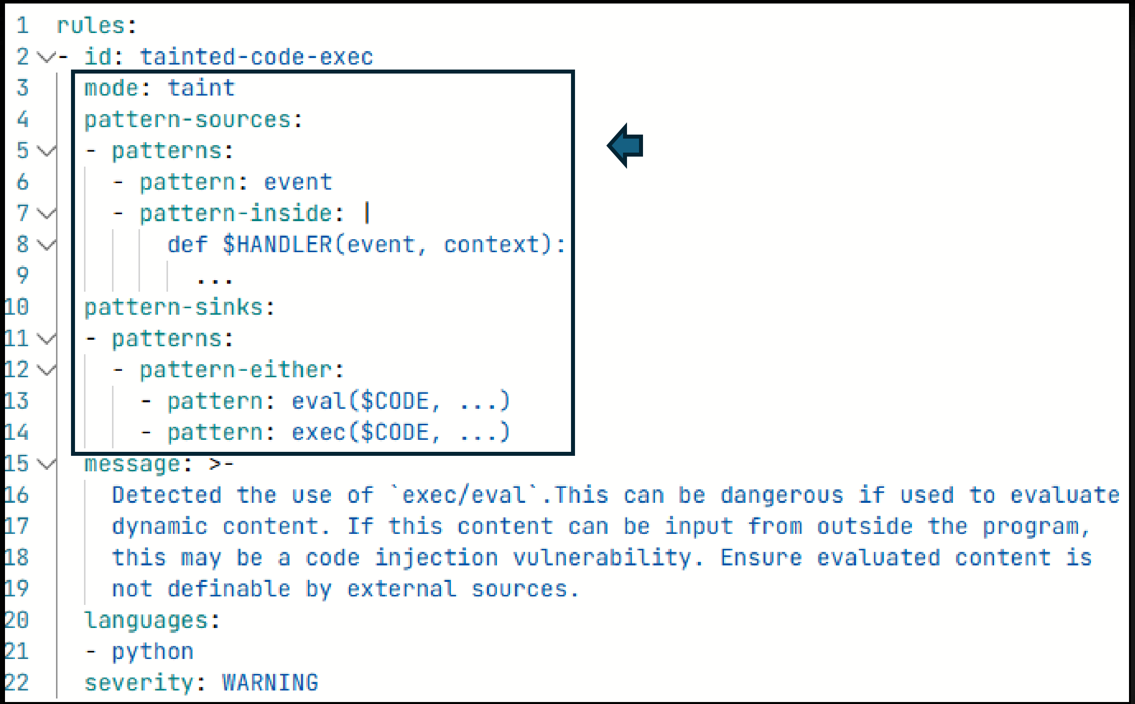
Advanced Semgrep Features

Overview of advanced Semgrep features: Taint mode, and editor integration.

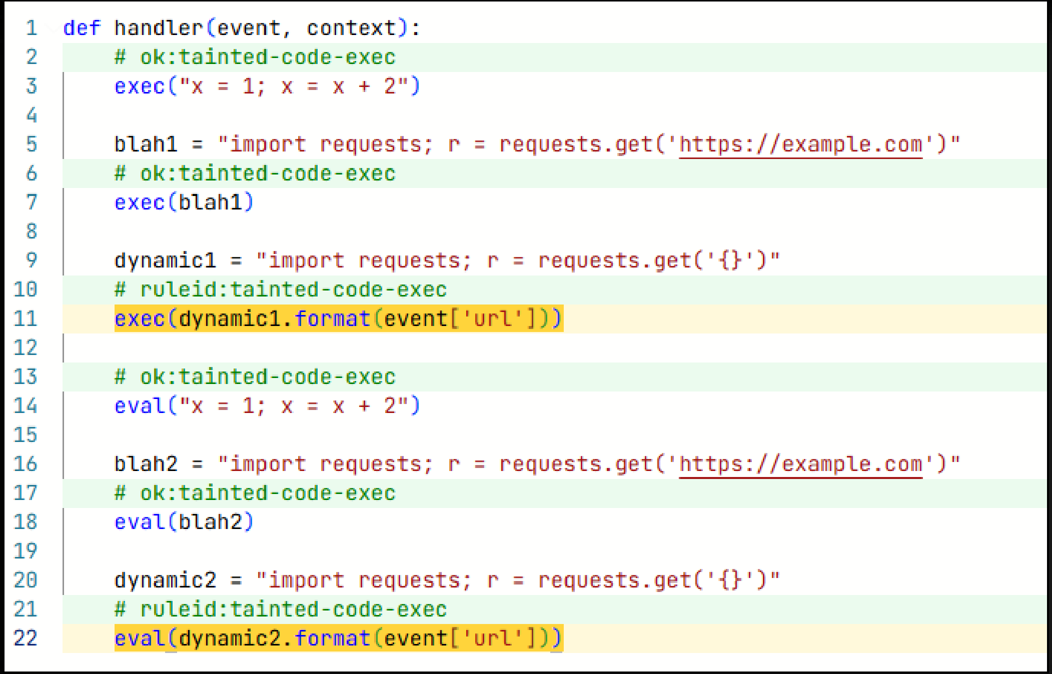
Leveraging Semgrep findings for LLM-based code analysis.

Taint Mode

- Tracks the flow of untrusted, or **tainted** data throughout the body of a function or method



```yaml
rules:
- id: tainted-code-exec
  mode: taint
  pattern-sources:
  - patterns:
    - pattern: event
    - pattern-inside: |
        def $HANDLER(event, context):
          ...
  pattern-sinks:
  - patterns:
    - pattern-either:
      - pattern: eval($CODE, ...)
      - pattern: exec($CODE, ...)
  message: >-
    Detected the use of `exec/eval`.This can be dangerous if used to evaluate
    dynamic content. If this content can be input from outside the program,
    this may be a code injection vulnerability. Ensure evaluated content is
    not definable by external sources.
  languages:
  - python
  severity: WARNING
```



```python
def handler(event, context):
    # ok:tainted-code-exec
    exec("x = 1; x = x + 2")

    blah1 = "import requests; r = requests.get('https://example.com')"
    # ok:tainted-code-exec
    exec(blah1)

    dynamic1 = "import requests; r = requests.get('{}')"
    # ruleid:tainted-code-exec
    exec(dynamic1.format(event['url']))

    # ok:tainted-code-exec
    eval("x = 1; x = x + 2")

    blah2 = "import requests; r = requests.get('https://example.com')"
    # ok:tainted-code-exec
    eval(blah2)

    dynamic2 = "import requests; r = requests.get('{}')"
    # ruleid:tainted-code-exec
    eval(dynamic2.format(event['url']))
```

Taint Mode | Semgrep

LLM-based Code Analysis

- SAST triage can be painful
  - Scale
  - Context
  - Confidence

- Leverage LLM capabilities
  - At scale
  - With context
  - Increased confidence

LLM-based Code Analysis

- SAST results can be exported via SARIF
- LLMs excel at (code) summarization
- Some LLMs have a ”JSON constrained” output guarantee
- LLMs exhibit understanding the JSON spec

Building it up

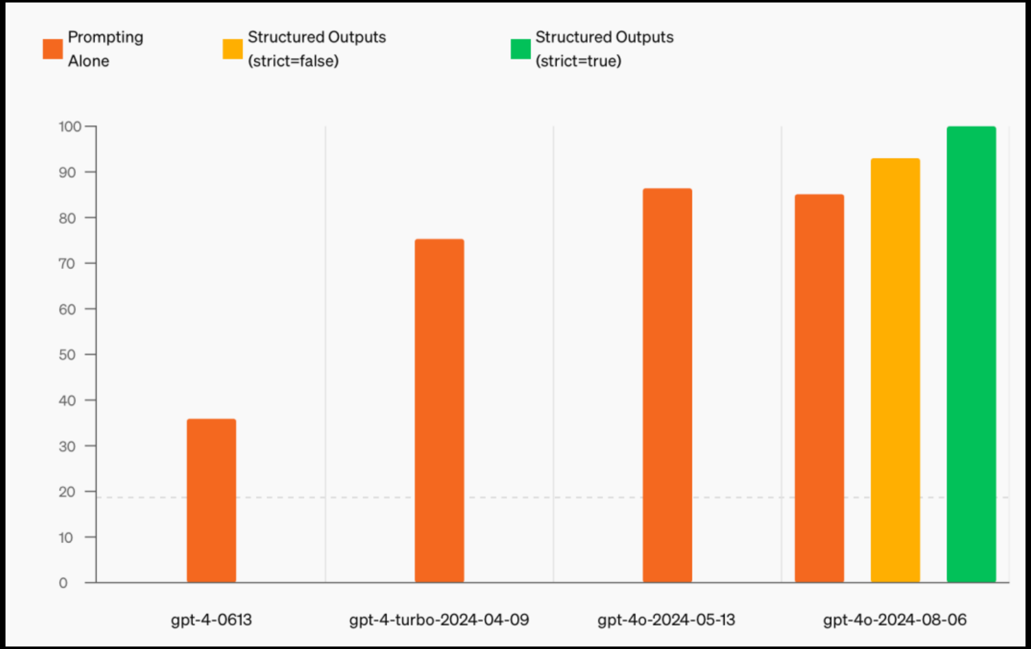
- SARIF offers interoperability between SAST tools
- SARIF is JSON
- Lots of language support to parse JSON

Building it up

- LLMs excel at summarization
- LLMs are also good at code generation
- LLMs need grounding and good prompts

Building it up

- LLMs love to talk
- LLMs can also be constrained to respond in certain manner, but with minimal guarantee of compliance
- Some LLMs have a “JSON constrained” output guarantee
  - Structured Output Mode



Building it up

- LLMs exhibit understanding the JSON spec

- Redux: JSON is great for parsing

- Constraining output to JSON allows for structured summarization

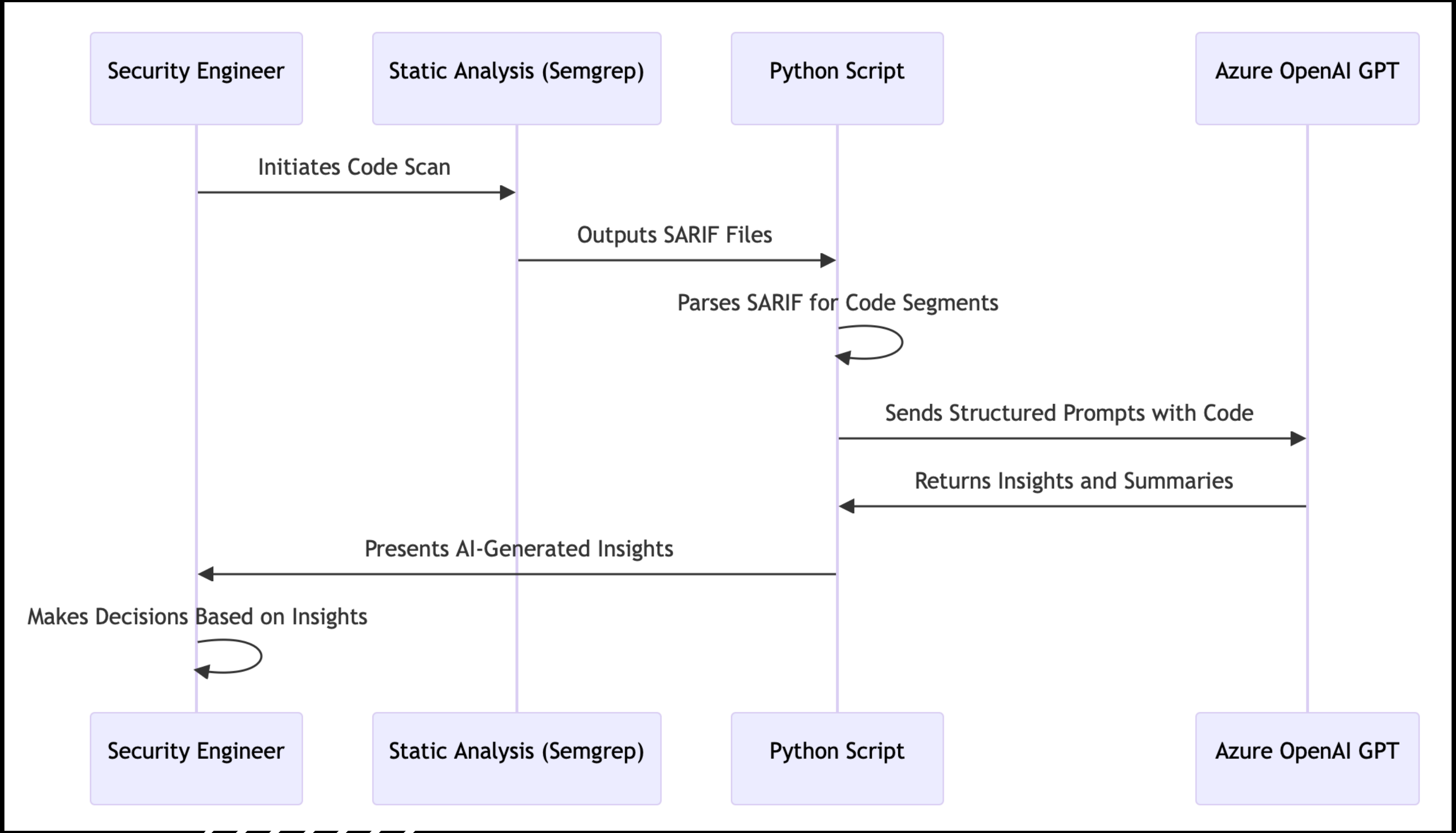
Building it up

- Prompting is part art/science
- Persona + Scenario + Output constraint + Input to analyze
- More “context” increases accuracy of LLM output
  - Custom rules to output full class body
  - Fetch file content from URL

Putting it together

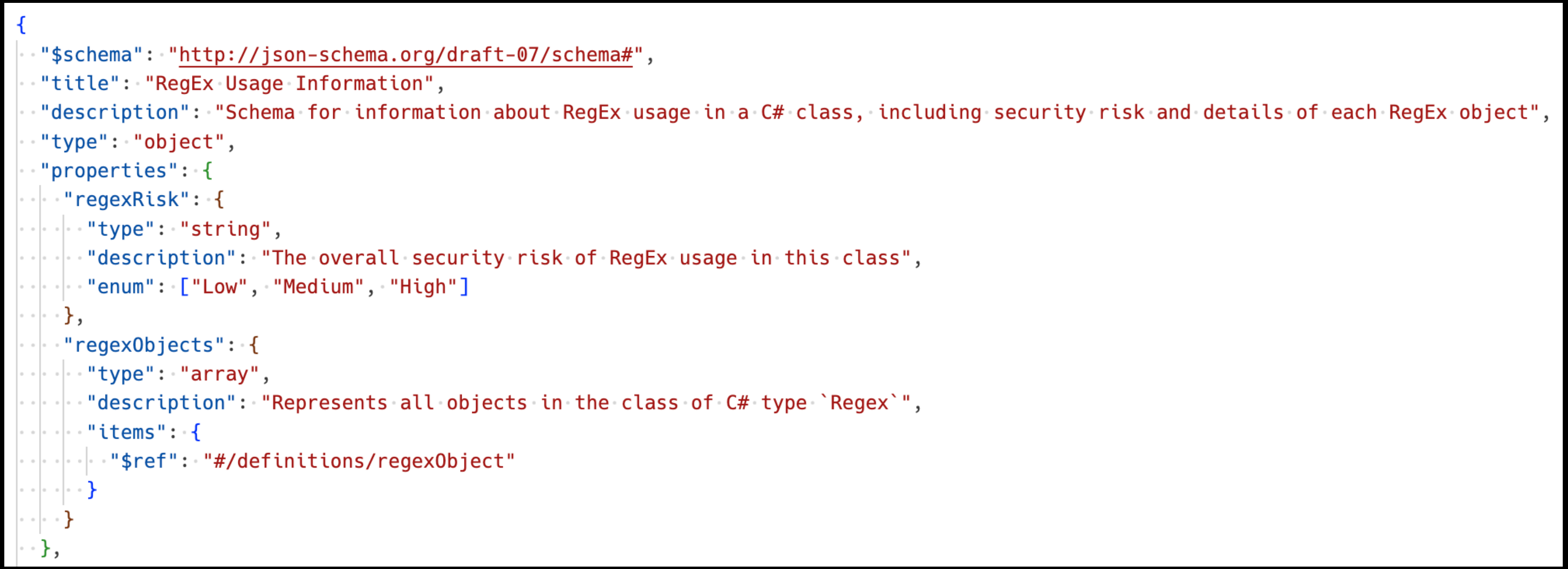
- Semgrep scan
  - Outputs SARIF
- SARIF (JSON) Parse
  - Extract code snippet
- Prompt GPT
  - System Prompt + JSON constraint on output + Code to analyze
- GPT Response
  - JSON output

Putting it together


Security Engineer
Static Analysis (Semgrep)
Python Script
Azure OpenAI GPT
Initiates Code Scan
Outputs SARIF Files
Parses SARIF for Code Segments
Sends Structured Prompts with Code
Returns Insights and Summaries
Presents AI-Generated Insights
Makes Decisions Based on Insights
Security Engineer
Static Analysis (Semgrep)
Python Script
Azure OpenAI GPT

Putting it together

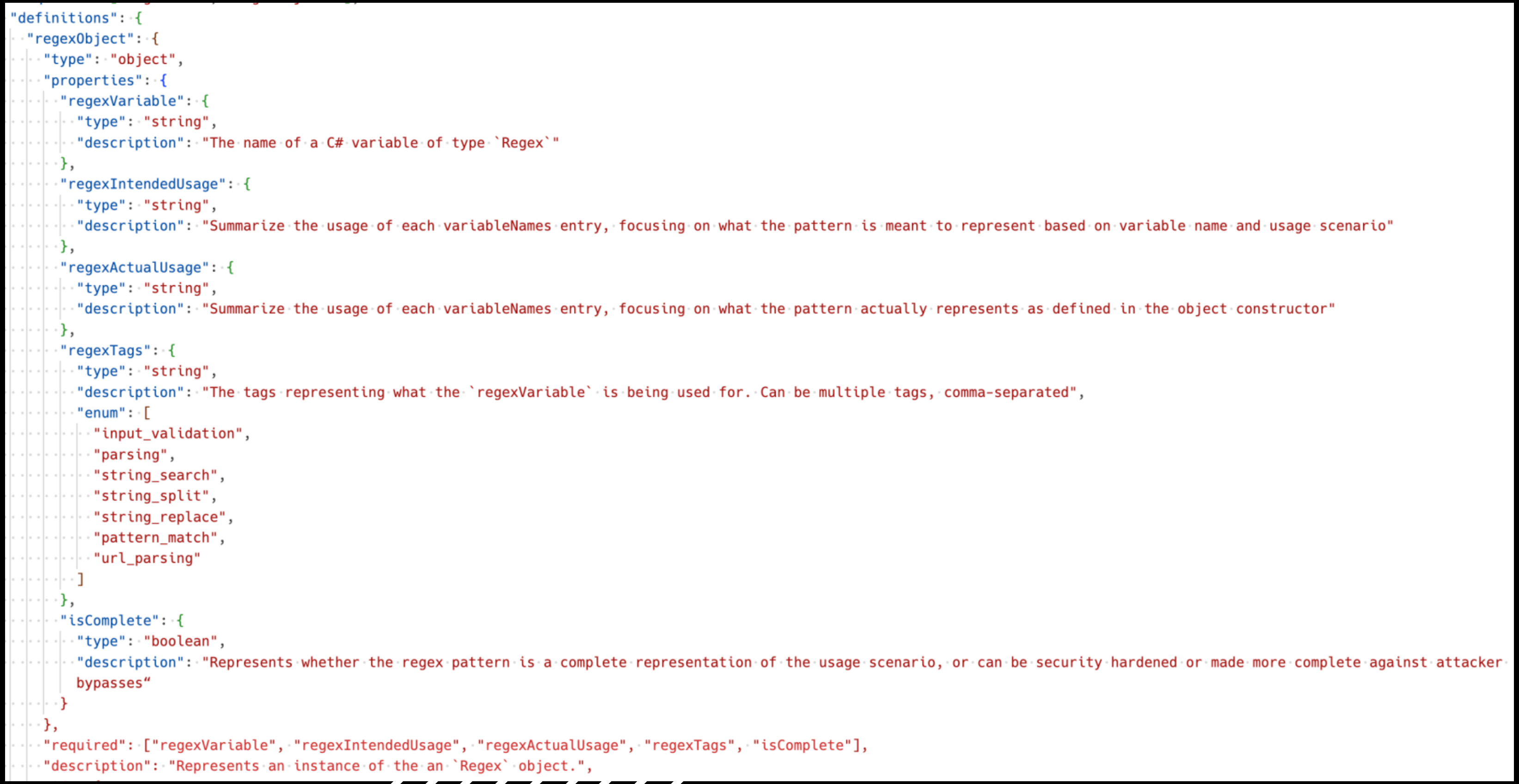
- Prompt
  - Basic



```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "RegEx Usage Information",
  "description": "Schema for information about RegEx usage in a C# class, including security risk and details of each RegEx object",
  "type": "object",
  "properties": {
    "regexRisk": {
      "type": "string",
      "description": "The overall security risk of RegEx usage in this class",
      "enum": ["Low", "Medium", "High"]
    },
    "regexObjects": {
      "type": "array",
      "description": "Represents all objects in the class of C# type `Regex`",
      "items": {
        "$ref": "#/definitions/regexObject"
      }
    }
  },
```

Putting it together

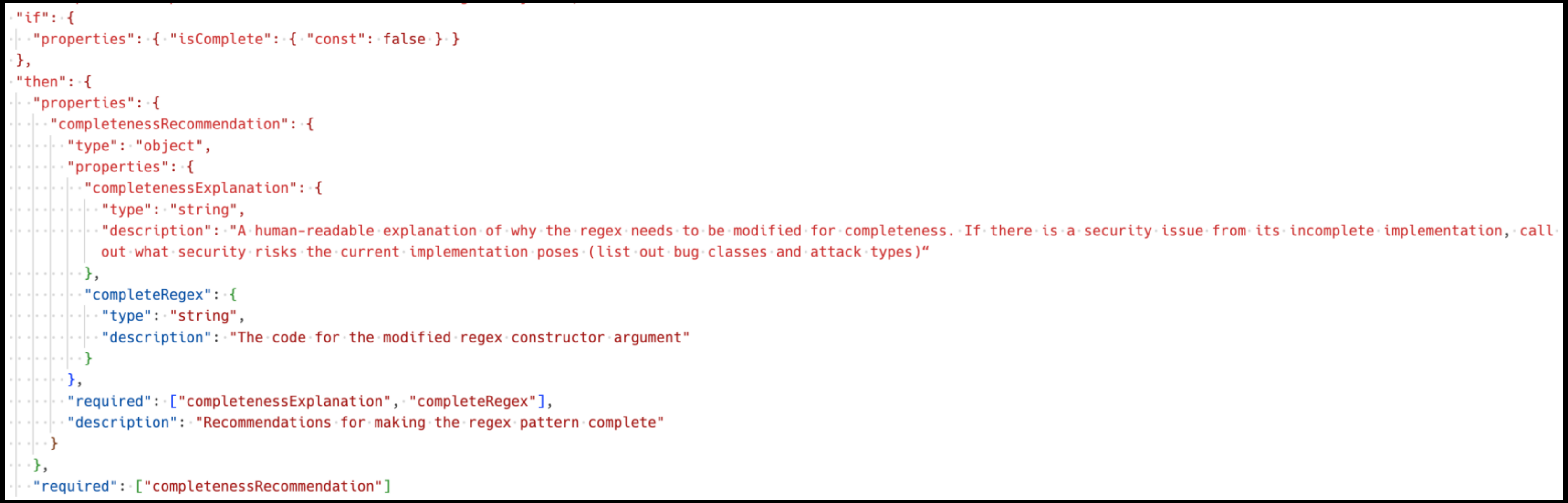
- Prompt
  - Better



```
"definitions": {
  "regexObject": {
    "type": "object",
    "properties": {
      "regexVariable": {
        "type": "string",
        "description": "The name of a C# variable of type `Regex`"
      },
      "regexIntendedUsage": {
        "type": "string",
        "description": "Summarize the usage of each variableNames entry, focusing on what the pattern is meant to represent based on variable name and usage scenario"
      },
      "regexActualUsage": {
        "type": "string",
        "description": "Summarize the usage of each variableNames entry, focusing on what the pattern actually represents as defined in the object constructor"
      },
      "regexTags": {
        "type": "string",
        "description": "The tags representing what the `regexVariable` is being used for. Can be multiple tags, comma-separated",
        "enum": [
          "input_validation",
          "parsing",
          "string_search",
          "string_split",
          "string_replace",
          "pattern_match",
          "url_parsing"
        ]
      },
      "isComplete": {
        "type": "boolean",
        "description": "Represents whether the regex pattern is a complete representation of the usage scenario, or can be security hardened or made more complete against attacker
        bypasses“
      }
    },
    "required": ["regexVariable", "regexIntendedUsage", "regexActualUsage", "regexTags", "isComplete"],
    "description": "Represents an instance of the an `Regex` object.",
```

Putting it together

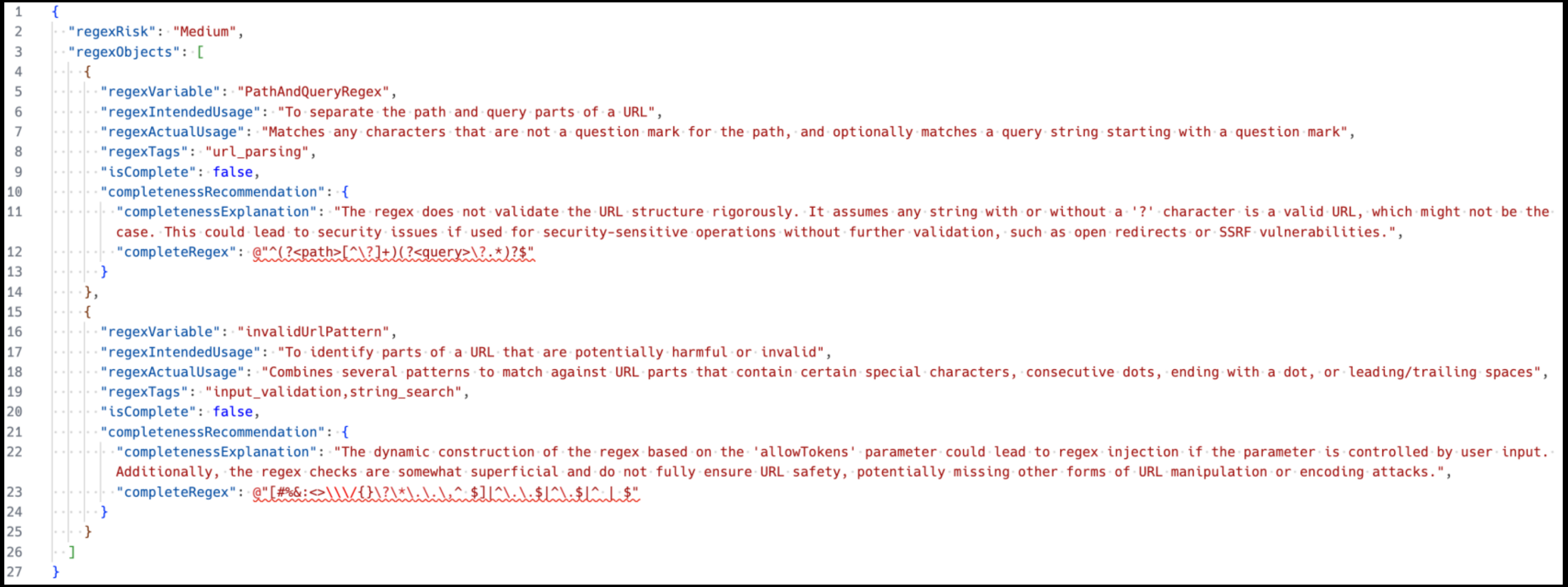
- Prompt
  - Even better (conditional sub-schemas)



```
"if": {
  "properties": { "isComplete": { "const": false } }
},
"then": {
  "properties": {
    "completenessRecommendation": {
      "type": "object",
      "properties": {
        "completenessExplanation": {
          "type": "string",
          "description": "A human-readable explanation of why the regex needs to be modified for completeness. If there is a security issue from its incomplete implementation, call
          out what security risks the current implementation poses (list out bug classes and attack types)“
        },
        "completeRegex": {
          "type": "string",
          "description": "The code for the modified regex constructor argument"
        }
      },
      "required": ["completenessExplanation", "completeRegex"],
      "description": "Recommendations for making the regex pattern complete"
    }
  },
  "required": ["completenessRecommendation"]
```

Putting it together

- Response



```json
{
  "regexRisk": "Medium",
  "regexObjects": [
    {
      "regexVariable": "PathAndQueryRegex",
      "regexIntendedUsage": "To separate the path and query parts of a URL",
      "regexActualUsage": "Matches any characters that are not a question mark for the path, and optionally matches a query string starting with a question mark",
      "regexTags": "url_parsing",
      "isComplete": false,
      "completenessRecommendation": {
        "completenessExplanation": "The regex does not validate the URL structure rigorously. It assumes any string with or without a '?' character is a valid URL, which might not be the case. This could lead to security issues if used for security-sensitive operations without further validation, such as open redirects or SSRF vulnerabilities.",
        "completeRegex": @"^(?<path>[^\?]+)(?<query>\?.*)?$"
      }
    },
    {
      "regexVariable": "invalidUrlPattern",
      "regexIntendedUsage": "To identify parts of a URL that are potentially harmful or invalid",
      "regexActualUsage": "Combines several patterns to match against URL parts that contain certain special characters, consecutive dots, ending with a dot, or leading/trailing spaces",
      "regexTags": "input_validation,string_search",
      "isComplete": false,
      "completenessRecommendation": {
        "completenessExplanation": "The dynamic construction of the regex based on the 'allowTokens' parameter could lead to regex injection if the parameter is controlled by user input. Additionally, the regex checks are somewhat superficial and do not fully ensure URL safety, potentially missing other forms of URL manipulation or encoding attacks.",
        "completeRegex": @"[#%&:<>\\\/{}\?\*\.\.\,^ $]|^\.\.$|^\.$|^ | $"
      }
    }
  ]
}
```

Initial Results

- Prioritized Triage
  - Riskiest <5%
- True Positives
  - >90%
- False Negatives
  - Harder to determine

Practical Exercises with Semgrep

MODULE 5

Practical Exercises with Semgrep

C#, TypeScript, Python

https://aka.ms/semgrep-workshop

Wrap-up

Wrap-up

- Recap of key takeaways from the workshop
- Learning resources and contributing to community

References

- Academy Courses
  - Semgrep 101
  - Secure Coding
  - Semgrep Custom Rules Level 1

- Semgrep Documentation
  - Quickstart
  - Local scans with Semgrep
  - Writing rules
  - Custom rule examples

Q&A