Analyse statique de code avec Semgrep

SSTIC 2023

Claudio Merloni (claudio@semgrep.com)
https://bit.ly/semgrep_sstic2023

Agenda

1. Introduction to Semgrep OSS
    a. Philosophy
    b. Architecture and features
    c. Getting started
    d. Alternatives
2. Finding vulnerabilities
3. Enforcing secure defaults
4. Exploring a code base
5. References

About me

Claudio Merloni
Security Research Manager @ Semgrep
claudio@semgrep.com
in claudiomerloni

Proudly stuck in static analysis since 2008
Formerly: Synopsys, Fortify

Semgrep OSS

Philosophy

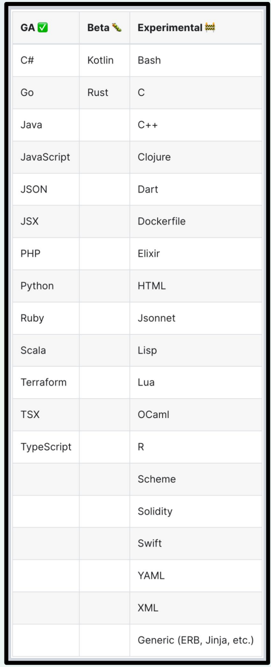
TL;DR

- Free & OSS
- Local & self-contained
- Runs everywhere, analyses everything
- User-friendly: get started using it and writing rules in minutes

https://semgrep.dev/docs/contributing/semgrep-philosophy/

Analysis architecture

Code

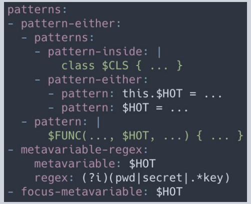


GA ✅	Beta 🐛	Experimental 🚧
C#	Kotlin	Bash
Go	Rust	C
Java		C++
JavaScript		Clojure
JSON		Dart
JSX		Dockerfile
PHP		Elixir
Python		HTML
Ruby		Jsonnet
Scala		Lisp
Terraform		Lua
TSX		OCaml
TypeScript		R
		Scheme
		Solidity
		Swift
		YAML
		XML
		Generic (ERB, Jinja, etc.)

Supported Languages

Rules

YAML patterns and metadata

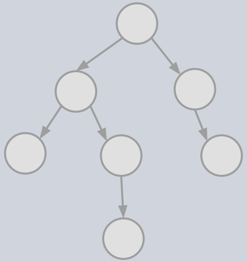


```yaml
patterns:
- pattern-either:
  - patterns:
    - pattern-inside: |
        class $CLS { ... }
    - pattern-either:
      - pattern: this.$HOT = ...
      - pattern: $HOT = ...
  - pattern: |
      $FUNC(..., $HOT, ...) { ... }
- metavariable-regex:
    metavariable: $HOT
    regex: (?i)(pwd|secret|.*key)
- focus-metavariable: $HOT
```

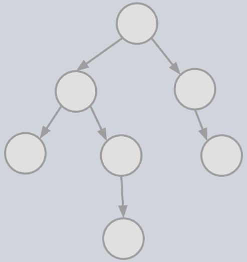
- Writing rules
- Rules repository
- Semgrep Playground
- Semgrep Registry

CST

tree-sitter parsers



AST/IL



Analysis

Search mode:
- Semantic-aware AST pattern matching
- Single file analysis
- Intraprocedural symbolic constant propagation
- Understanding of types and external symbols
- Autofix

Taint mode:
- Extension of search mode
- Intraprocedural taint propagation
- Extends syntax with support for sources, sinks, sanitizers and propagators
- Taint labels and composition

Extract mode:
- Extract and analyse code embedded in a different language

Join mode:
- Multi file analysis
- Join results based on metavariables and file paths

Getting started

- *Prerequisites*: Python, WSL on Windows
- *Installation*
  - brew install semgrep
  - pip install semgrep (or pipx install semgrep)
- *Usage*
  - semgrep --config=<rules> <my target folder>
- *Alternatively*
  - docker run --rm -v "${PWD}:/src" returntocorp/semgrep semgrep --config=<rules>

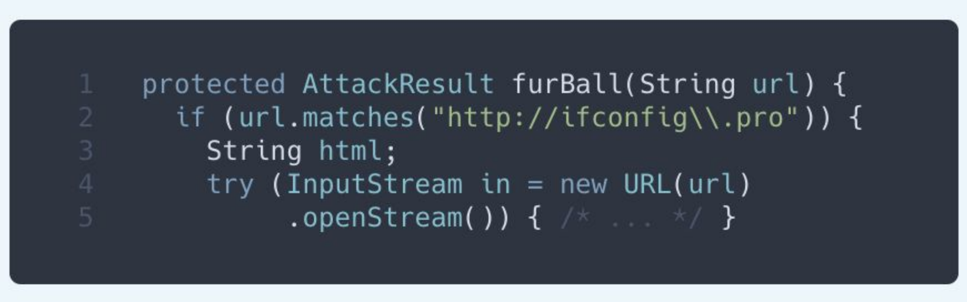
- Can be used in CLI, pre-commit, CI workflows, etc.
- See https://semgrep.dev/docs/getting-started/

Alternatives

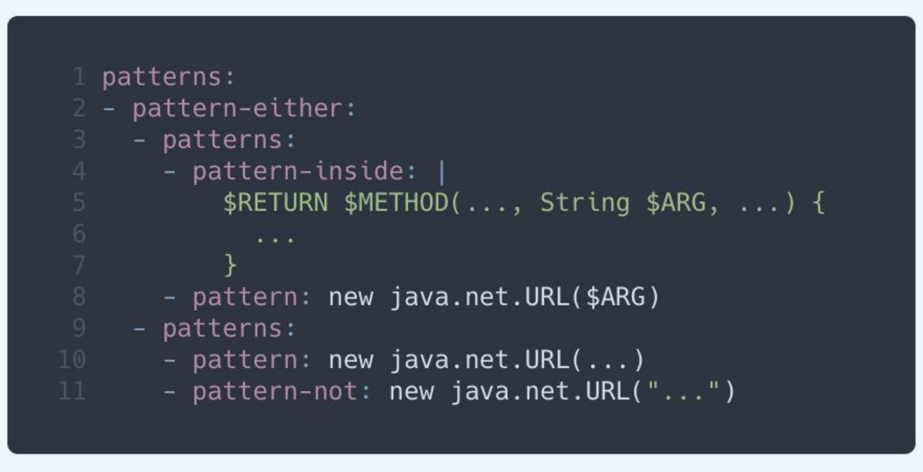
- *CodeQL*
  - Engine is not OSS
  - Open to contributions to queries and libraries
  - Free to use only on public GitHub repositories
  - Supports C, C++, C#, Go, Java, Kotlin, JS/TS, Python, Ruby
  - Requires full "buildable" code base
  - Query syntax has rather high learning curve
- *Weggli*
  - Written in Rust, doesn't depend on something like a Python installation
  - Parsing based on tree-sitter
  - Focused on C and C++
  - Doesn't require buildable code base
  - Patterns are very close to C/C++ code
  - Stay here to hear more about it from Kevin Denis!
- ... and many others! *Gosec, Brakeman, Bandit, Gitleaks, ...*

Finding vulnerabilities

Audit for SSRF



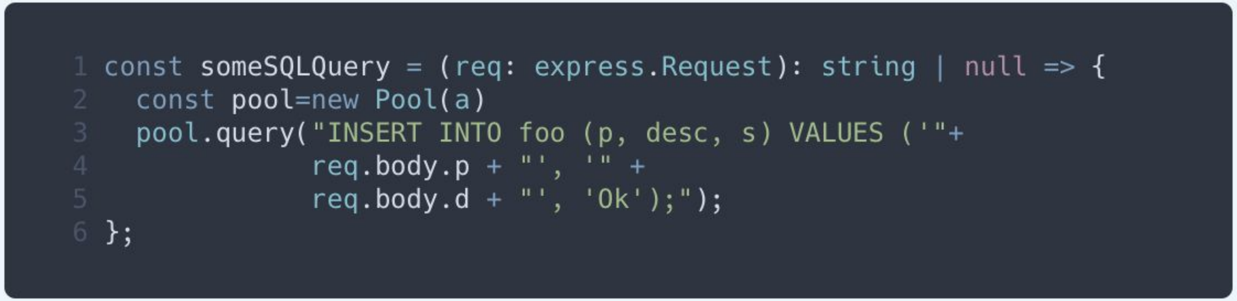
```java
protected AttackResult furBall(String url) {
  if (url.matches("http://ifconfig\\.pro")) {
    String html;
    try (InputStream in = new URL(url)
            .openStream()) { /* ... */ }
```



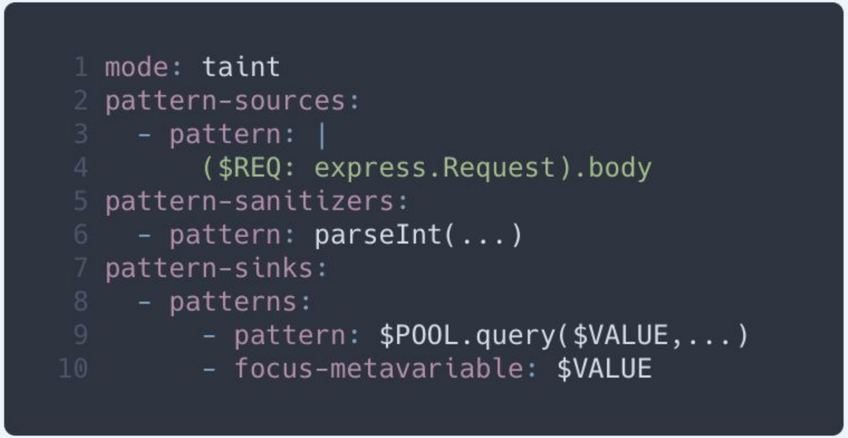
```yaml
patterns:
- pattern-either:
  - patterns:
    - pattern-inside: |
        $RETURN $METHOD(..., String $ARG, ...) {
          ...
        }
    - pattern: new java.net.URL($ARG)
  - patterns:
    - pattern: new java.net.URL(...)
    - pattern-not: new java.net.URL("...")
```

Try it out: https://semgrep.dev/s/J4jw
Similar vulnerability, different audit technique: https://blog.doyensec.com/2023/03/16/ssrf-remediation-bypass.html

Hunting for SQLi



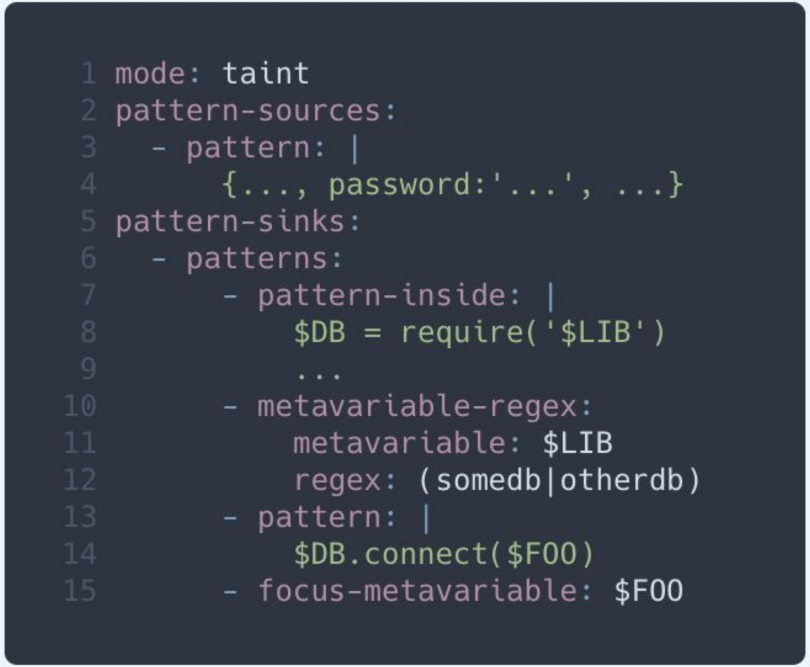
```
const someSQLQuery = (req: express.Request): string | null => {
  const pool=new Pool(a)
  pool.query("INSERT INTO foo (p, desc, s) VALUES ('"+
             req.body.p + "', '" +
             req.body.d + "', 'Ok');");
};
```



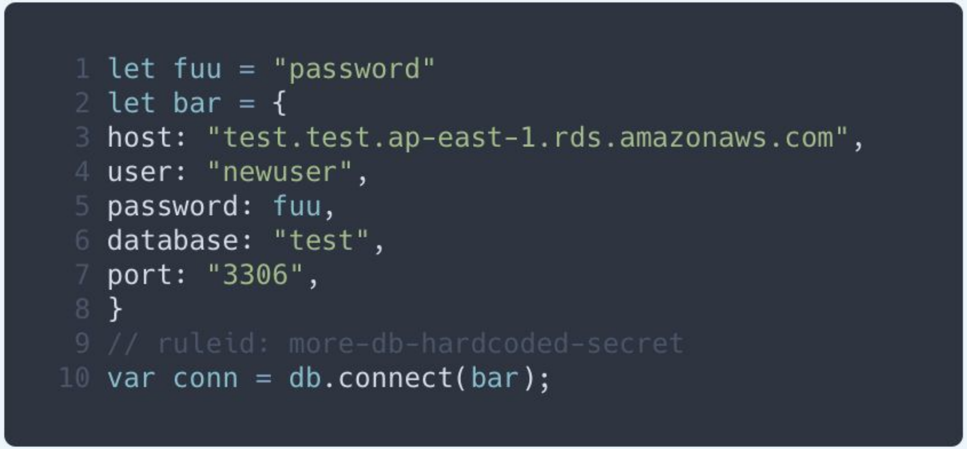
```
mode: taint
pattern-sources:
  - pattern: |
      ($REQ: express.Request).body
pattern-sanitizers:
  - pattern: parseInt(...)
pattern-sinks:
  - patterns:
      - pattern: $POOL.query($VALUE,...)
      - focus-metavariable: $VALUE
```

Try it out: https://semgrep.dev/playground/s/2oyj

Hunting for hardcoded secrets



```yaml
mode: taint
pattern-sources:
  - pattern: |
      {..., password:'...', ...}
pattern-sinks:
  - patterns:
      - pattern-inside: |
          $DB = require('$LIB')
          ...
      - metavariable-regex:
          metavariable: $LIB
          regex: (somedb|otherdb)
      - pattern: |
          $DB.connect($FOO)
      - focus-metavariable: $FOO
```

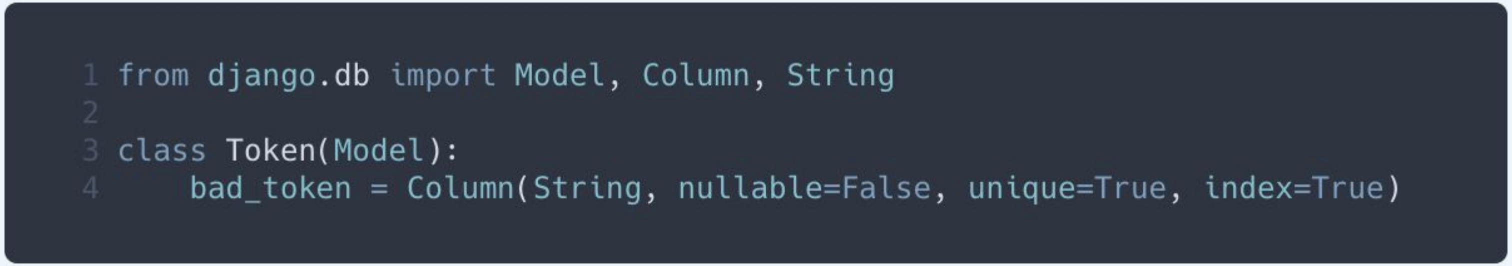


```javascript
let fuu = "password"
let bar = {
host: "test.test.ap-east-1.rds.amazonaws.com",
user: "newuser",
password: fuu,
database: "test",
port: "3306",
}
// ruleid: more-db-hardcoded-secret
var conn = db.connect(bar);
```

Try it out: https://semgrep.dev/s/vypW

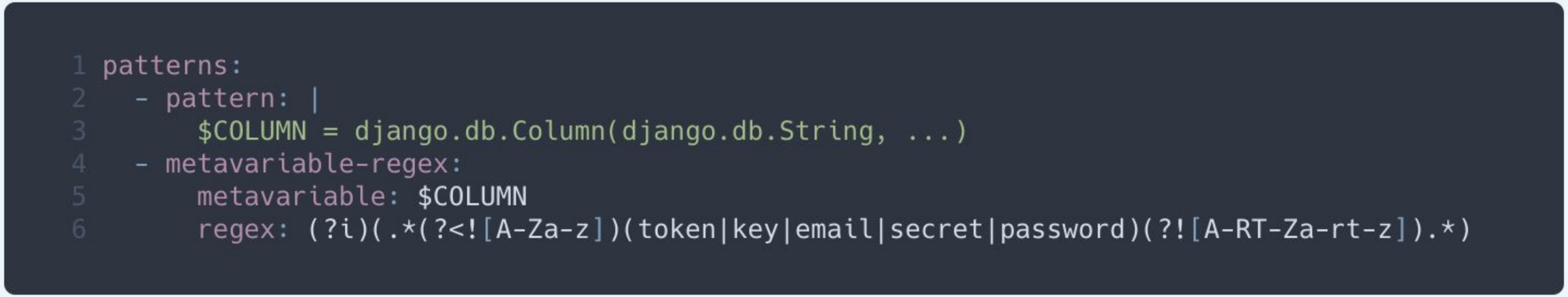
Enforcing secure defaults

Stop leaking sensitive data



```python
from django.db import Model, Column, String

class Token(Model):
    bad_token = Column(String, nullable=False, unique=True, index=True)
```



```yaml
patterns:
  - pattern: |
      $COLUMN = django.db.Column(django.db.String, ...)
  - metavariable-regex:
      metavariable: $COLUMN
      regex: (?i)(.*(?<![A-Za-z])(token|key|email|secret|password)(?![A-RT-Za-rt-z]).*)
```

Try it out: https://semgrep.dev/s/0klB

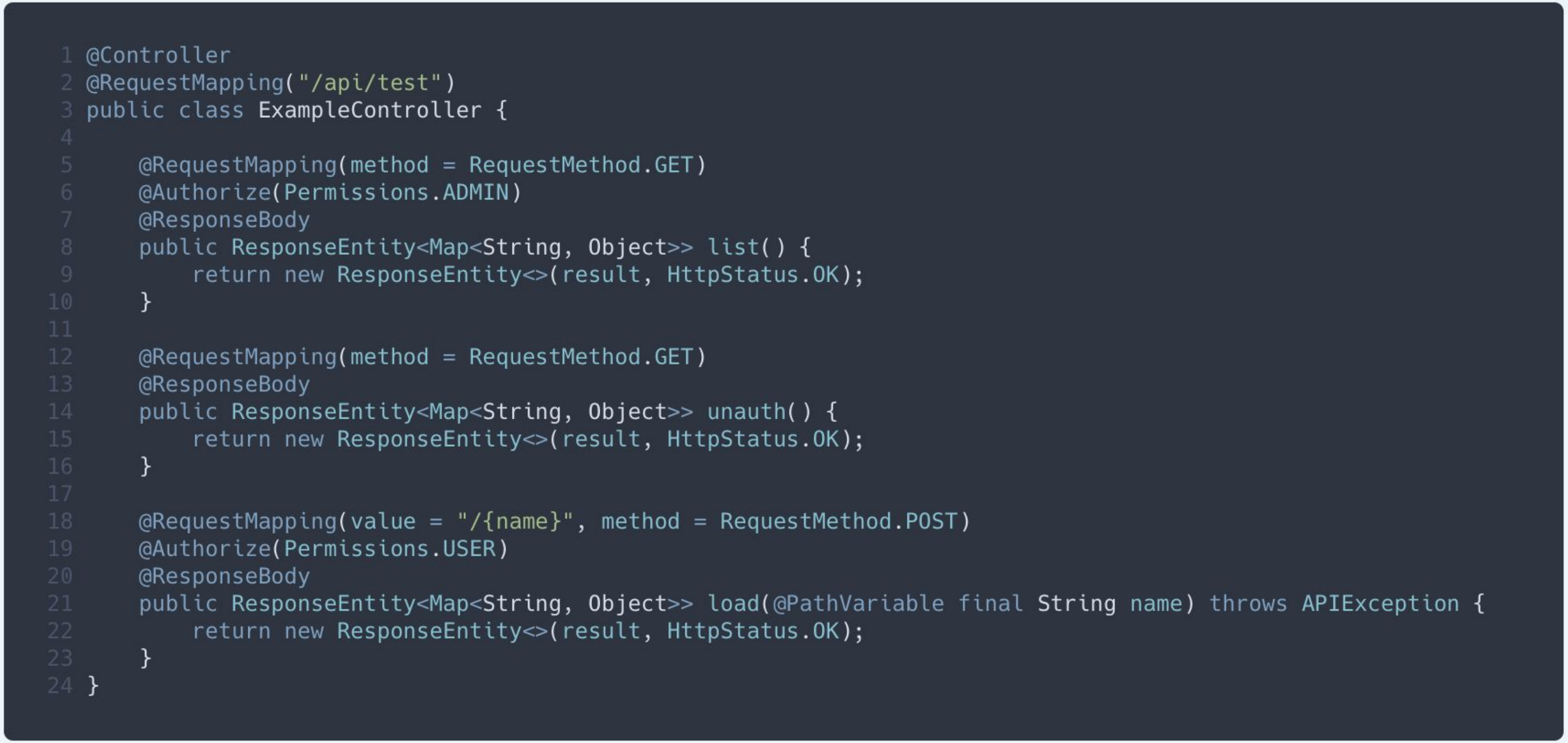
And code conventions in general!

- Validate format of structured data files (JSON, YAML, etc.)
  - We use it for our own Semgrep rules!
- Enforcing naming conventions (classes, API endpoints, etc.)
- Usage of forbidden/unsafe libraries and APIs

Example: https://www.fabianzeindl.com/posts/business-information-server

Exploring a code base

Discovering web application routes



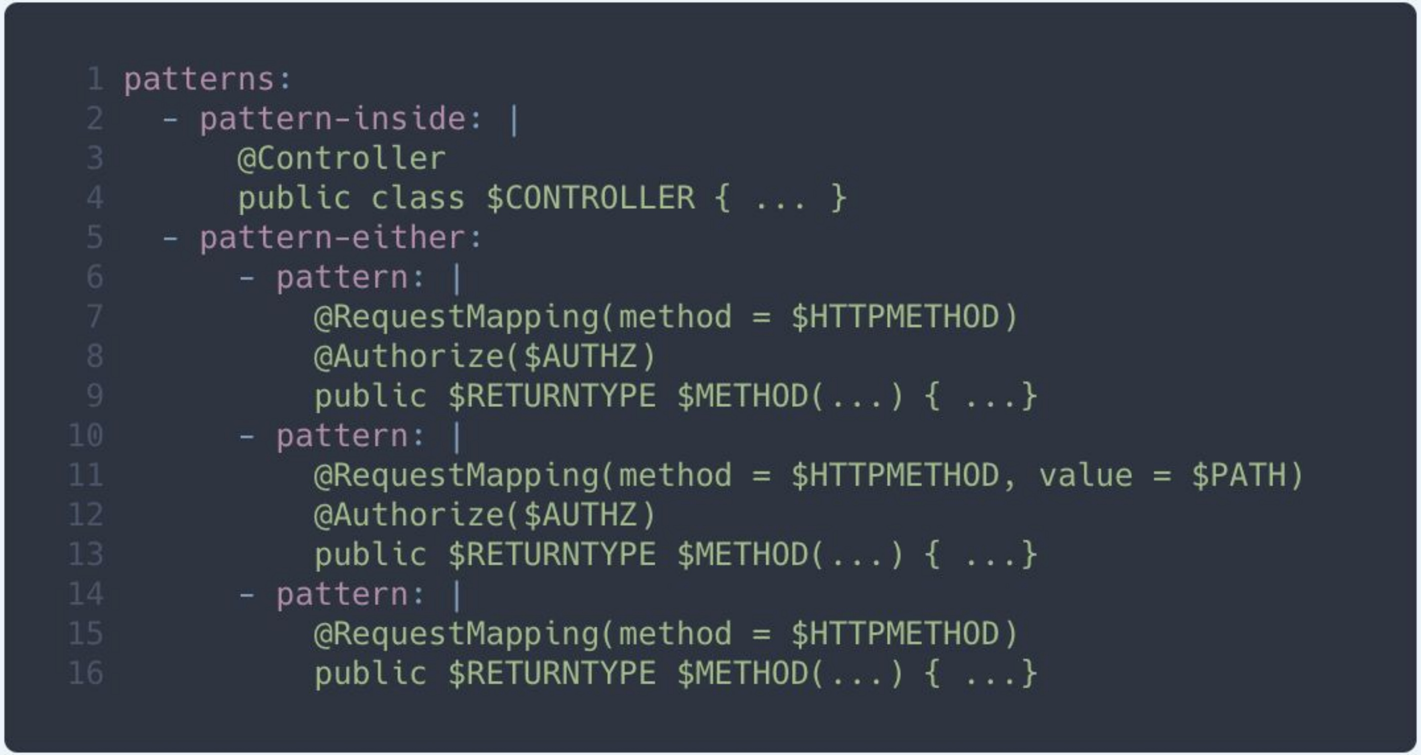
```java
@Controller
@RequestMapping("/api/test")
public class ExampleController {

    @RequestMapping(method = RequestMethod.GET)
    @Authorize(Permissions.ADMIN)
    @ResponseBody
    public ResponseEntity<Map<String, Object>> list() {
        return new ResponseEntity<>(result, HttpStatus.OK);
    }

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    public ResponseEntity<Map<String, Object>> unauth() {
        return new ResponseEntity<>(result, HttpStatus.OK);
    }

    @RequestMapping(value = "/{name}", method = RequestMethod.POST)
    @Authorize(Permissions.USER)
    @ResponseBody
    public ResponseEntity<Map<String, Object>> load(@PathVariable final String name) throws APIException {
        return new ResponseEntity<>(result, HttpStatus.OK);
    }
}
```

Discovering web application routes



```
patterns:
  - pattern-inside: |
      @Controller
      public class $CONTROLLER { ... }
  - pattern-either:
      - pattern: |
          @RequestMapping(method = $HTTPMETHOD)
          @Authorize($AUTHZ)
          public $RETURNTYPE $METHOD(...) { ...}
      - pattern: |
          @RequestMapping(method = $HTTPMETHOD, value = $PATH)
          @Authorize($AUTHZ)
          public $RETURNTYPE $METHOD(...) { ...}
      - pattern: |
          @RequestMapping(method = $HTTPMETHOD)
          public $RETURNTYPE $METHOD(...) { ...}
```

Try it out: https://semgrep.dev/s/WdqL

References

References

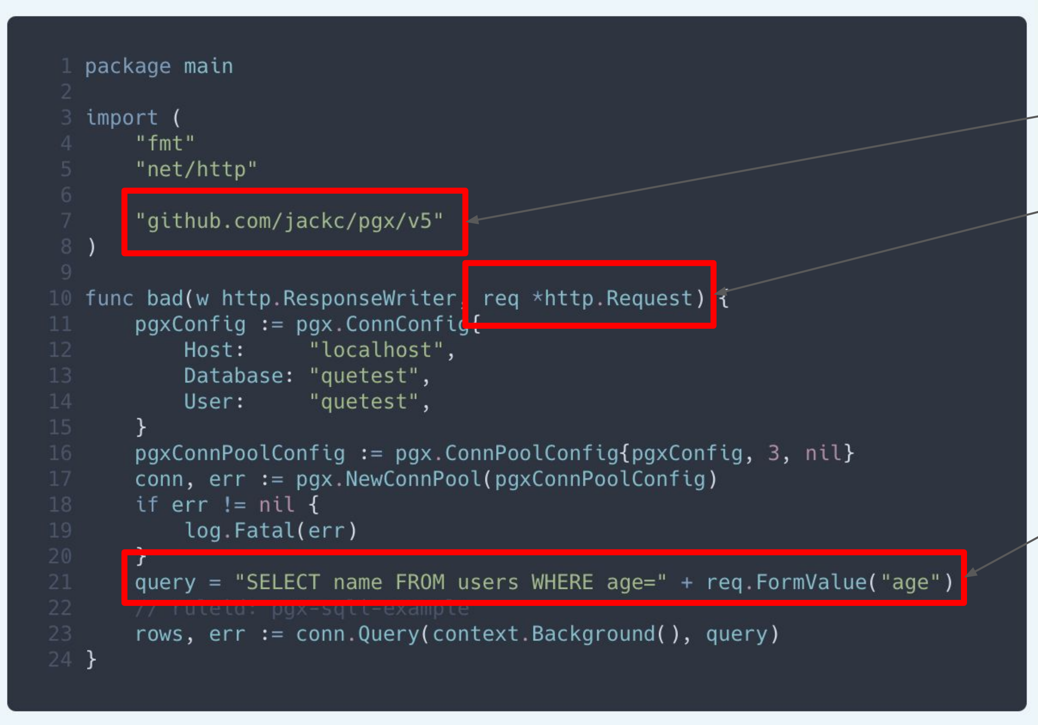
- Semgrep source code: https://github.com/returntocorp/semgrep
- Registry: https://semgrep.dev/explore
  - Community rules repository: https://github.com/returntocorp/semgrep-rules
- Rule writing tutorial: https://semgrep.dev/learn
- Docs: https://semgrep.dev/docs/
- Community Slack: https://r2c.dev/slack

- https://blog.trailofbits.com/2021/11/08/discovering-goroutine-leaks-with-semgrep/
- https://blog.includesecurity.com/2021/07/customizing-semgrep-rules-for-flask-django/
- https://notsosecure.com/semgrep-practical-introduction
- https://blog.aquia.io/blog/2022-02-18-semgrep-cdk/
- https://blog.doyensec.com/2023/03/16/ssrf-remediation-bypass.html
- https://www.fabianzeindl.com/posts/business-information-server

Merci!

More examples

SQL Injection with *taint labels* - The code



```go
package main

import (
    "fmt"
    "net/http"

    "github.com/jackc/pgx/v5"
)

func bad(w http.ResponseWriter, req *http.Request) {
    pgxConfig := pgx.ConnConfig{
        Host:     "localhost",
        Database: "quetest",
        User:     "quetest",
    }
    pgxConnPoolConfig := pgx.ConnPoolConfig{pgxConfig, 3, nil}
    conn, err := pgx.NewConnPool(pgxConnPoolConfig)
    if err != nil {
        log.Fatal(err)
    }
    query = "SELECT name FROM users WHERE age=" + req.FormValue("age")
    // ruleid: pgx-sqli-example
    rows, err := conn.Query(context.Background(), query)
}
```

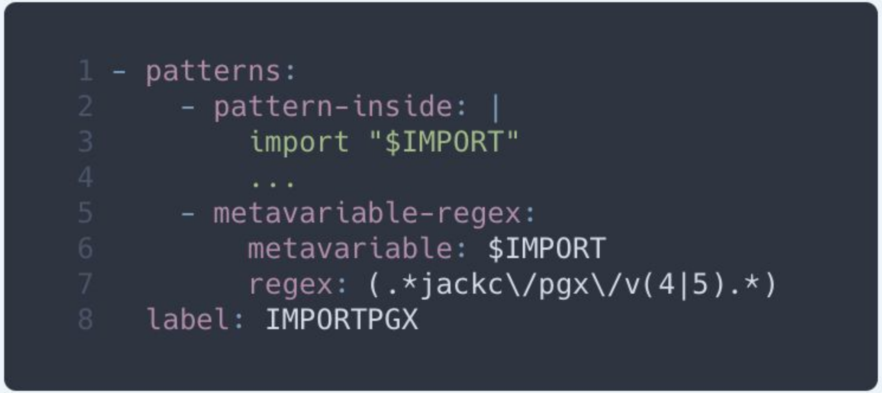
Focus on a specific library

We know where user data comes from

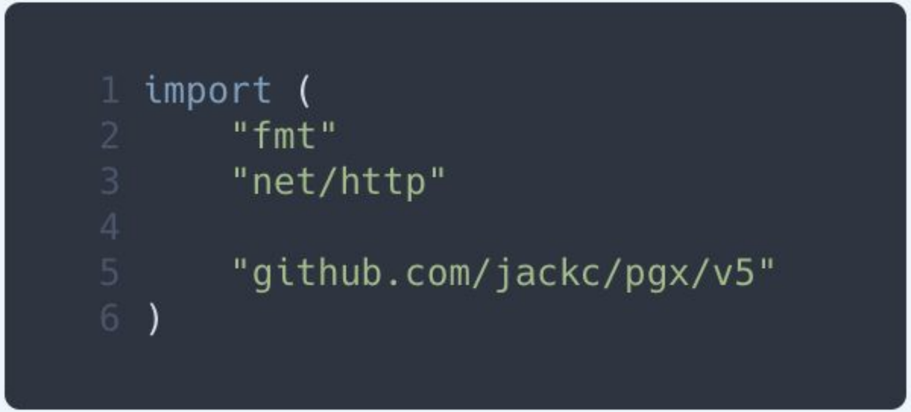
The usual suspect ...

Try it out: https://semgrep.dev/s/kID1

SQL Injection with *taint labels* - Database library



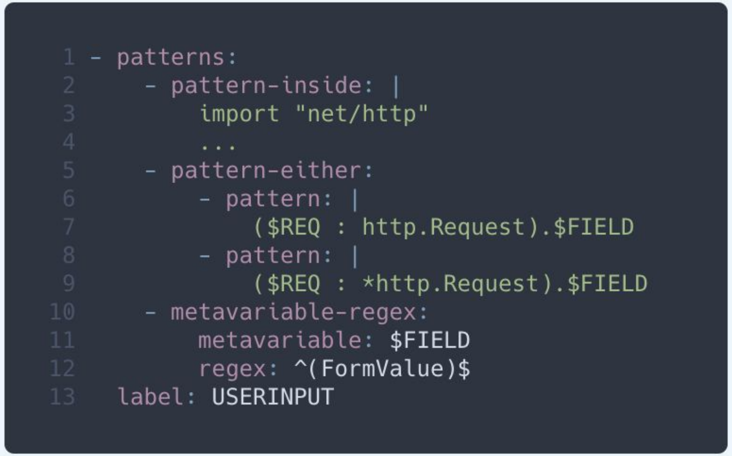
```yaml
- patterns:
  - pattern-inside: |
      import "$IMPORT"
      ...
  - metavariable-regex:
      metavariable: $IMPORT
      regex: (.*jackc\/pgx\/v(4|5).*)
  label: IMPORTPGX
```



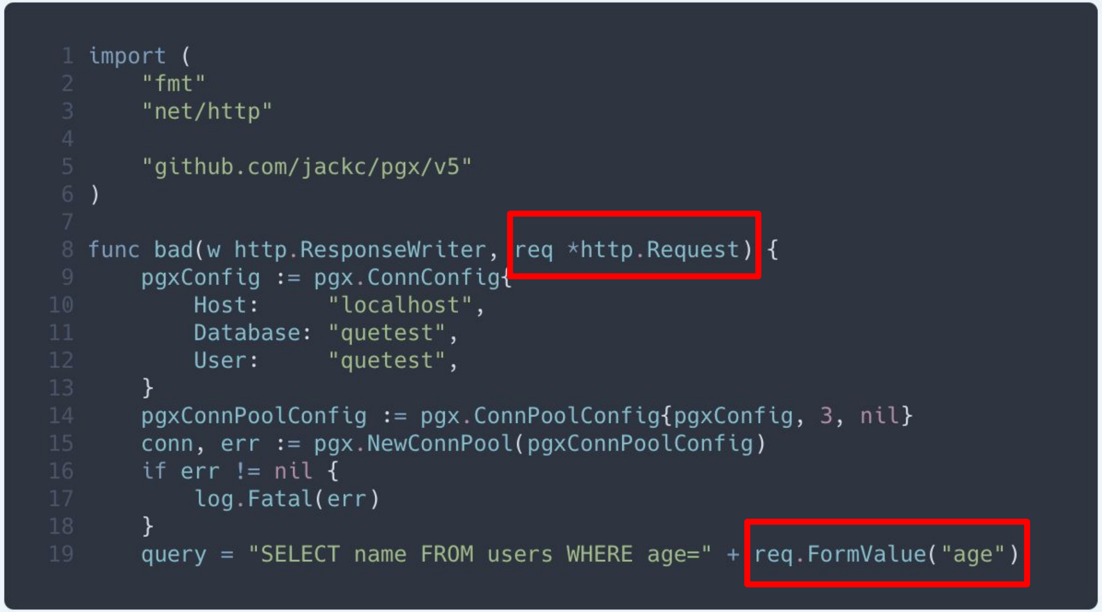
```go
import (
	"fmt"
	"net/http"

	"github.com/jackc/pgx/v5"
)
```

SQL Injection with *taint labels* - User input



```
- patterns:
    - pattern-inside: |
        import "net/http"
        ...
    - pattern-either:
        - pattern: |
            ($REQ : http.Request).$FIELD
        - pattern: |
            ($REQ : *http.Request).$FIELD
    - metavariable-regex:
        metavariable: $FIELD
        regex: ^(FormValue)$
  label: USERINPUT
```

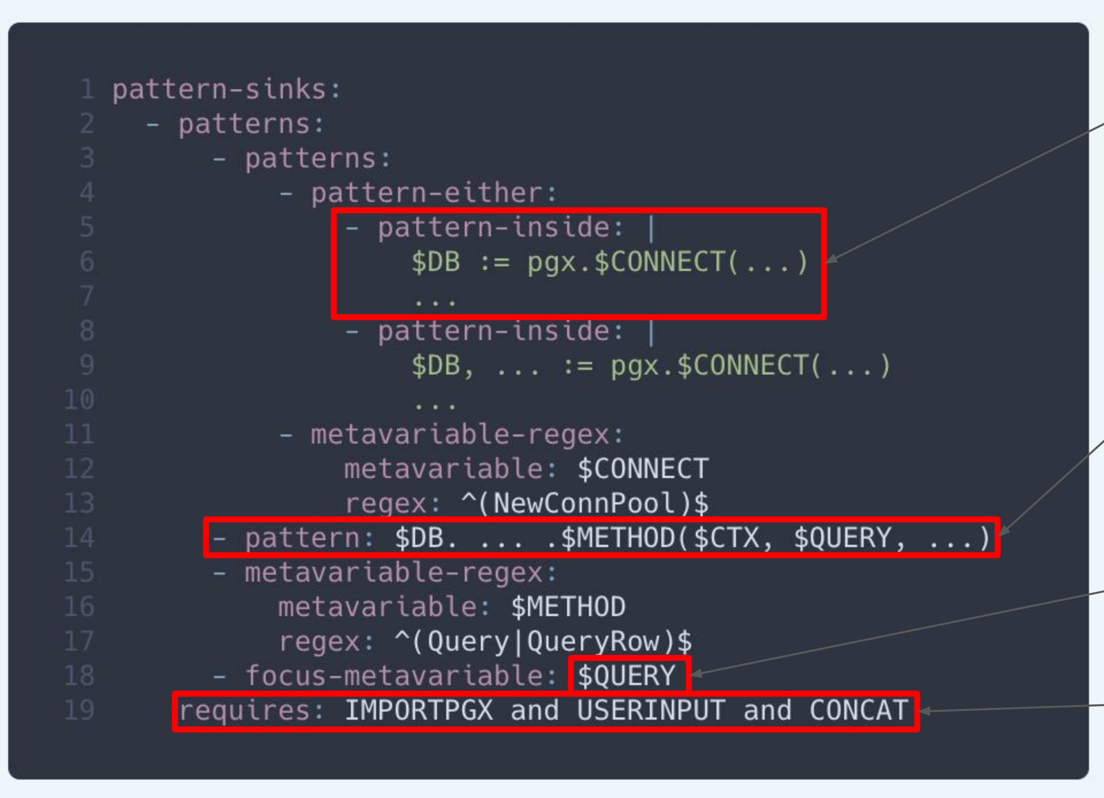


```
import (
    "fmt"
    "net/http"

    "github.com/jackc/pgx/v5"
)

func bad(w http.ResponseWriter, req *http.Request) {
    pgxConfig := pgx.ConnConfig{
        Host:     "localhost",
        Database: "quetest",
        User:     "quetest",
    }
    pgxConnPoolConfig := pgx.ConnPoolConfig{pgxConfig, 3, nil}
    conn, err := pgx.NewConnPool(pgxConnPoolConfig)
    if err != nil {
        log.Fatal(err)
    }
    query = "SELECT name FROM users WHERE age=" + req.FormValue("age")
```

SQL Injection with *taint labels* - Putting it together



```
pattern-sinks:
  - patterns:
      - patterns:
          - pattern-either:
              - pattern-inside: |
                  $DB := pgx.$CONNECT(...)
                  ...
              - pattern-inside: |
                  $DB, ... := pgx.$CONNECT(...)
                  ...
          - metavariable-regex:
              metavariable: $CONNECT
              regex: ^(NewConnPool)$
      - pattern: $DB. ... .$METHOD($CTX, $QUERY, ...)
      - metavariable-regex:
          metavariable: $METHOD
          regex: ^(Query|QueryRow)$
      - focus-metavariable: $QUERY
    requires: IMPORTPGX and USERINPUT and CONCAT
```

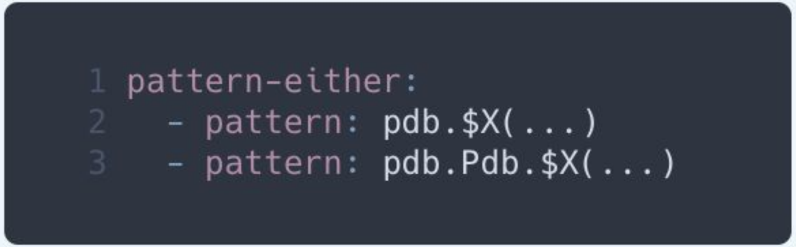
Focus on a block of code and capture the database connection

Call on a method of the captured variable

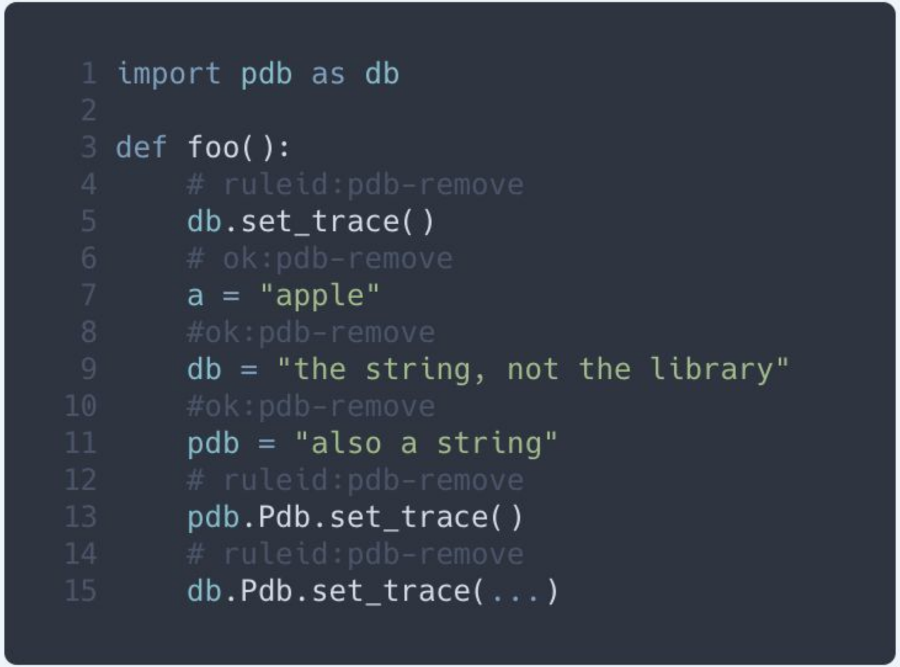
The query is our sink ...

... only if source constraints are met!

Forgotten debugging code



```yaml
pattern-either:
  - pattern: pdb.$X(...)
  - pattern: pdb.Pdb.$X(...)
```



```python
import pdb as db

def foo():
    # ruleid:pdb-remove
    db.set_trace()
    # ok:pdb-remove
    a = "apple"
    #ok:pdb-remove
    db = "the string, not the library"
    #ok:pdb-remove
    pdb = "also a string"
    # ruleid:pdb-remove
    pdb.Pdb.set_trace()
    # ruleid:pdb-remove
    db.Pdb.set_trace(...)
```

Try it out: https://semgrep.dev/r/python.lang.correctness.pdb.pdb-remove