KITCTF 02.03.2023 - CodeQL Workshop

Setup instructions: https://gh.io/nc-2023-setup

`FROM code SELECT vulnerability`

- Why static analysis?
    - **Static analysis**: Finding problems in code without executing it.
    - ⇒ Find vulnerabilities even in rarely executed code.
- Why CodeQL?
    - Precisely model (vulnerability) patterns.
    - Extendable, **open-source** queries.
    - Powerful data flow analysis.
    - Reusable & shareable queries.
    - Scaling: Find bugs in 10s or 1000s of programs.

Workshop Goals

- Get to know CodeQL.
- Write your first query.
- Avoid common pitfalls.
- Learn tips and tricks.

Workshop Format

- Interactive workshop!
- We first explain concepts and for each concept we provide small exercises.
- There can be multiple solutions for the same exercises.
  This is expected!
- If you have *any* questions, feel free to interrupt us and ask!

Checkpoint: Setup

- Did you follow the setup instructions on https://gh.io/nc-2023-setup?
- If something does not work, now is the time to fix it!

Introduction

What is CodeQL

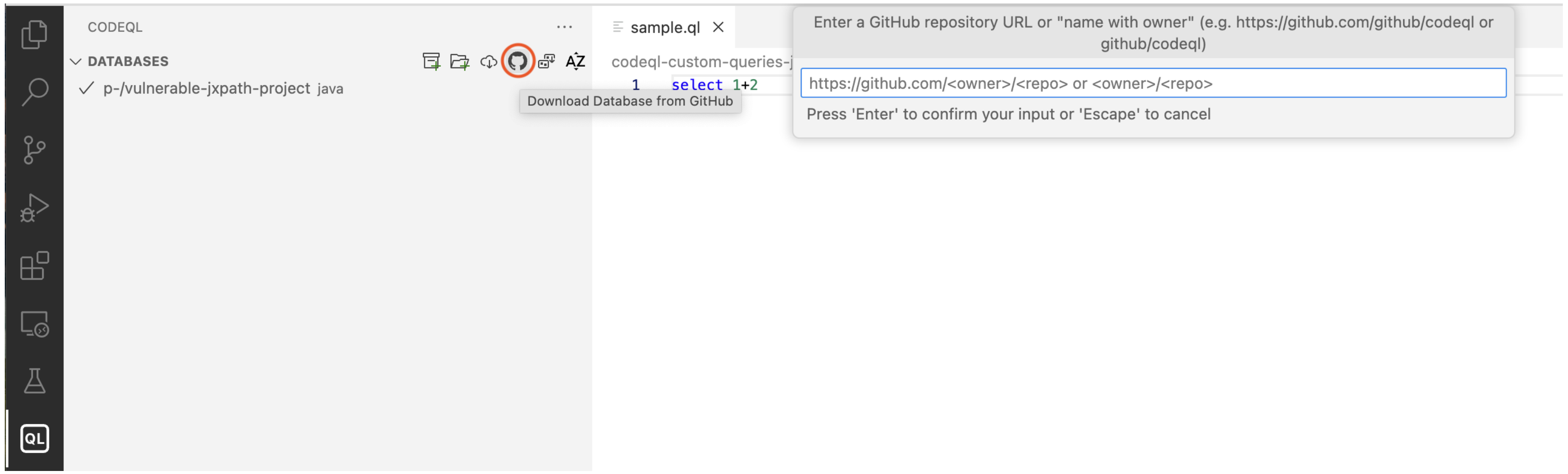
- CodeQL is ...
  - a static analysis tool.
  - a *logic-based* and *object-oriented* programming language.
  - a tool to turn *code into data*.
- Allows us to use logic to reason about *code as data*.
- Supports Java/Kotlin, C#, C/C++, JavaScript/TypeScript, Python, Go, Ruby, and Swift (Beta).

What is a CodeQL Database?

- Collection of facts about the code:
    - Abstract syntax tree + control flow graph + data flow + ...
- Contains a copy of your source code.

  ⇒ everything needed is contained in the database.

How to Get a Database I

- Prebuilt databases:
  - Provided by GitHub.
  - Available via the REST API: https://docs.github.com/en/rest/code-scanning?apiVersion=2022-11-28#get-a-codeql-database-for-a-repository
  - Or directly in the VS Code extension:
  - 

CODEQL
DATABASES
p-/vulnerable-jxpath-project java
Download Database from GitHub
sample.ql
codeql-custom-queries-j

```
select 1+2
```

Enter a GitHub repository URL or "name with owner" (e.g. https://github.com/github/codeql or github/codeql)

https://github.com/<owner>/<repo> or <owner>/<repo>

Press 'Enter' to confirm your input or 'Escape' to cancel

How to Get a Database II

- Self-built databases:
  - Using `codeql` cli.
  - More information: https://docs.github.com/en/code-security/codeql-cli/using-the-codeql-cli/creating-codeql-databases

CodeQL Query Structure

- A **query** has three parts:
  - `from <type> variable1, <type> variable2, ...`: define values we are working on.
  - `where <formula holds>`: filter values.
  - `select <alertLocation>, "message"`: create an alert at the location using the given message.

    (this is basically SQL but written upside-down to enhance autocompletion!)

CodeQL Query Structure (continued)

- A query **file** uses the **.ql** extension and contains a **query**. It can also contain imports, classes, and predicates.
- A query **library** uses the **.qll** extension and **does not** contain a **query**. It can also contain imports, classes, and predicates.

Example: CodeQL Query Structure

- `import java`: imports the java query library; makes classes and predicates available
- `from Class javaStringClass`: from **all** elements that represent Java classes
- `where javaStringClass.hasQualifiedName("java.lang", "String")`: only get the class that represents the Java String class.
- `select javaStringClass, "This is the Java String class."`: create an alert.

Examples

```
select "Hello KITCTF"
```

Examples

```
from int year
where year = 2023
select "Hello KITCTF " + year
```

Examples

```
from string greeting, string target
where greeting = "Hello" and target = "everyone"
select greeting + " " + target + "!"
```

Examples

```
from Class javaStringClass
where javaStringClass.hasQualifiedName("java.lang", "String")
select javaStringClass, "This is the Java String class."
```

Building Blocks

Building Blocks for Java

- The following building blocks are **specific to Java**.
- But the **concepts** are transferrable to the other languages supported (JavaScript, Ruby, Go, C, and others) because what changes are mostly **keywords.**

Program Elements

- Represent named program elements:
- Types (`Type`).
- Methods (`Method`).
- Constructors (`Constructor`).
- Variables (`Variable`).
- Common superclass: `Element` .
- Helpful member predicates:
  - `Element.contains(Element e)` : Holds if this element transitively contains `e` .
  - `Element.hasName(string name)` : Holds if this element has the name `name` .

Program Elements: Variables

- `Variable` represents variables in the Java sense:
    - `Field` represents a Java field.
    - `LocalVariableDecl` represents a local variable.
    - `Parameter` represents a parameter of a method or constructor.
- Helpful member predicates:
    - `VarAccess Variable.getAnAccess()`: Gets an access to this variable.
    - `Expr Variable.getAnAssignedValue()`: Gets an expression that is assigned to this variable.

Program Elements: Types

- Class `Type` represents different kinds of Java types:
  - `PrimitiveType` represents a primitive type: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`.
  - `RefType` represents a reference (non-primitive) type; it has several subclasses:
    - `Class` represents a Java class.
    - `Interface` represents a Java interface.
    - `Enum`Type represents a Java `enum` type.
    - `Array` represents a Java array type.

Program Elements: Types (Continued)

- Helpful member predicates:
  - `RefType.getAMethod`: Gets a method declared in this type.
  - `RefType.getASubtype`: Gets a direct subtype of this type.
  - `RefType.getASupertype`: Gets a direct supertype of this type.
  - `RefType.getAnAncestor`: Gets a direct or indirect supertype of this type, including itself.
  - `RefType.hasQualifiedName(string package, string type)`: Holds if this type is declared in a specified package with the specified name.

Program Elements: Methods

- `Method`: Represents Java methods, for example, `public void foo() {}`
- Helpful member predicates:
  - `Method.hasName(string name)`: holds if this method has the specified name.
  - `Method.getAPossibleImplementation`: get a method that could be called when this method is called.
  - `Method.getAReference`: get a reference to an expression that invokes this method.
- `MethodAccess`: Represents an invocation of a method with a list of arguments; example: `foo()`

Program Elements: Constructors

- `Constructor`: Represents Java constructors, for example, `public Bar() {}`
- `ConstructorCall`: Represents constructor calls via `new`, `this()`, `super()`.

-
  ```
  new Bar(); // <-- constructor call
  ```

-
  ```
  public Bar() {
    this("foo"); // <-- constructor call
  }
  public Bar(String string) {}
  ```

Program Elements: Calls and Callables

- `Callable`: Common super class for `Method` and `Constructor`.
- `Call`: Common super class for `MethodAccess` and `ConstructorCall`.

Exercises: Program Elements

Exercise 1: Find all variables named "pathAdjusted" of type String.

Hints:

- Get the type of a variable using the `getType()` method.
- Check if a type is of type String by using `instanceof TypeString`

Exercise 2: Find all methods that have a parameter of type String and are in a sub package of "seclab.testprojects.jxpathvuln."

Hint: Match the start of a String like this:

```
matches("seclab.testprojects.jxpathvuln.%")
```

(Optional) Exercise 3: Find all static fields whose name starts with "ERROR".

Abstract Syntax Tree Nodes

- Abstract syntax trees (ASTs) represent the structure of program code.
- Java's AST has two types of nodes:
  - Statements: Modeled via the `Stmt` CodeQL class.
  - Expressions: Modeled via the `Expr` CodeQL class.

Full list: https://codeql.github.com/docs/codeql-language-guides/abstract-syntax-tree-classes-for-working-with-java-programs/

Abstract Syntax Tree Nodes (Continued)

- Helpful member predicates:
  - `Expr.getAChildExpr` returns a sub-expression of a given expression.
  - `Stmt.getAChild` returns a statement or expression that is nested directly inside a given statement.
  - `Expr.getParent` and `Stmt.getParent` return the parent node of an AST node.

Abstract Syntax Tree Nodes: Statements

Statement syntax	CodeQL class
`Expr ;`	ExprStmt
`{ Stmt ... }`	BlockStmt
`if ( Expr ) Stmt else`	IfStmt
`while ( Expr ) Stmt`	WhileStmt
`return Expr ;`	ReturnStmt

Abstract Syntax Tree Nodes: Expressions

Expression syntax	CodeQL class
Literals: `true`, `23`, `"Hello"`, ...	BooleanLiteral, IntegerLiteral, StringLiteral, ...
Unary expressions: `Expr++`, `--Expr`, `!Expr`, ...	PostIncExpr, PreDecExpr, LogNotExpr, ...
Binary expressions: `Expr * Expr`, `Expr && Expr`, `Expr < Expr`, ...	MulExpr, AndLogicalExpr, LTExpr, ...

Abstract Syntax Tree Nodes: Expressions (Continued)

Expression syntax	CodeQL class
Assignment expressions: `Expr = Expr`, `Expr += Expr`, ...	AssignExpr, AssignAddExpr, ...
Accesses: `x`, `obj.field`, `array[0]`, `obj.method()`, ...	VarAccess, FieldAccess, ArrayAccess, MethodAccess, ...

Exercises: Abstract Syntax Tree Nodes

Exercise 4: Find all accesses to methods with name `getValue` that do not take place inside callables with names starting with `get`.

Exercise 5: Find all calls to `getValue` that use a constant string as the first argument.

Hints:

- Use `StringLiteral` to check for instance of constant strings.
- CodeQL indices are zero-based.

Predicates

- They establish a relationship between its parameters by means of a formula.
- A predicate represents the **set of tuples** that satisfy its formula.
    - A database table if you will.
- A predicate **holds** when its formula evaluates to true on the input.

Predicates (Continued)

- Example:

```
predicate isSmallEvenNumber(int i) {
  i % 2 = 0 and // is even?
  i in [1..10] // is small?
}
```

- Does `isSmallEvenNumber(12)` hold? No, because `12 in [1..10]` is false.
- Does `isSmallEvenNumber(10)` hold? Yes, because `10 % 2 = 0` is true and `10 in [1..10]` is true.

Predicates (Continued 2)

- They are similar to functions but the analogy will become weird if you push it too much.
- Since there is no state, just a formula, you can evaluate anything in QL and get the results.
    - **This is an incredible feature and should be used extensively to understand the bits and bolts of more complex queries.**

Built-in Predicates

- `any()`: a predicate that always holds, "true".
- `none()`: a predicate that never holds, "false".
- `string.matches`: holds when the receiver matches the argument in the same way as the LIKE operator in SQL. `_` matches a single character and `%` matches any sequence of characters.
- `string.toLowerCase`: returns the receiver with all letters converted to lower case.
- `string.regexpMatch`: holds when the receiver matches the argument as a regex.

Full list: https://codeql.github.com/docs/ql-language-reference/ql-language-specification/#built-ins

Classes

- Describe a set of values that share a characteristic.
- Every class needs a super type.
- *Characteristic predicate* determines which values are part of the class.
- Member predicates allow adding useful "methods".

```
class [ClassName] extends [SuperType] {
  [ClassName]() { // <- characteristic predicate
    // constrain the values that [ClassName] contains
  }
  predicate memberPredicate() {
  }
}
```

Classes: Example

- Characteristic: All calls to the methods named `exec`.
- Calls to methods are represented by `MethodAccess`.
  ⇒ super type is `MethodAccess`
- "Take all calls to methods, but only those named `exec`. Give those values the name `ExecMethodAccess`"

```
class ExecMethodAccess extends MethodAccess {
  ExecMethodAccess() { // <- characteristic predicate
    this.getMethod().hasName("exec")
  }
}
```

Explicit Type Checks

- `instanceof`: Used to check whether a value is of a certain type.
- Using `instanceof` is **completely** natural in CodeQL.

```
from MethodAccess ma
where ma instanceof ExecMethodAccess
select ma, "call to exec"
```

Casts

- Allow constraining expressions to a type.
    - Postfix style cast: `x.(Foo)`
    - Prefix style cast: `((Foo)x)`
    - `x` is now constrained to be of type `Foo`.
- Can use both styles, but prefix casts are rarely used.

Casts: Java vs. CodeQL

- Casts in Java **can** throw an error/exception.
- Casts in CodeQL **do not** throw an error/exception.

  ⇒ Allow us to **chain** predicates easily.

```
predicate foo(MethodAccess ma) {
  ma.(ExecMethodAccess).cmdArgument().(AddExpr).
  getRightOperand().getType().(RefType).hasQualifiedName("java.lang", "String")
}
```

Quantifiers

- There are three of them (`exists`, `forall`, and `forex`) but the most used is `exists`.
  - `exists(<variable declarations> | <formula>)`
  - Reads as "there is an X such as".
  - Holds if the variables declared satisfy the formula.
  - Allows us to introduce temporary variables.

More information: https://codeql.github.com/docs/ql-language-reference/formulas/#quantified-formulas

Data Flow Analysis

Data Flow Analysis

- Allows us to reason about the propagation of data.
- Allows us to answer questions like these:
  - Does this expression reach X?
  - Where does this expression reach in the program?
- Fundamental in more complex queries.

More Information: https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-java/

Data Flow Analysis: Flavors

- Local
  - Follows propagation within a single function.
  - Precise and relatively cheap.
  - `DataFlow::localFlow`, `DataFlow::localExprFlow`
- Global
  - Follows propagation across functions.
  - Less precise and computationally expensive.
  - `DataFlow::Configuration`, `Configuration::hasFlow`

AST Nodes vs. Data Flow Nodes

- AST nodes reflect the **syntactic structure** of the program.
- Classes: `Expr` , `MethodAccess` , `VarAccess` , ...
- Data flow nodes model the way data flows through the program at runtime.
- Nodes in the data flow graph represent **semantic elements** that carry values at runtime.
- Class: `DataFlow::Node`
- We often translate between AST and data flow nodes:
  `Expr DataFlow::Node.asExpr()` : Gets the expression corresponding to this node, if any.

Local Data Flow Analysis

- Follows propagation within a single function.
- Import `semmle.code.java.dataflow.DataFlow` and then use `DataFlow::localExprFlow`.
- `DataFlow::localExprFlow(Expr e1, Expr e2)`: holds if there is flow from `e1` to `e2`.

Exercises: Local Data Flow

Exercise 6: The query from exercise 5 found calls like this `getValue("string")`, but does not find constructs like this:

```
String pathSelector = "selector";
getValue(pathSelector);
```

Hints:

- `import semmle.code.java.dataflow.DataFlow`
- Use `DataFlow::localExprFlow` to track the flow of string literals to `getValue`.

Global Data Flow Analysis

- Follows propagation across functions.
- To make the problem tractable we need to define a **source** and a **sink**.
- To use, import `semmle.code.java.dataflow.DataFlow` and extend `DataFlow::Configuration`.
  - Override `Configuration::isSource(DataFlow::Node src)`
  - Override `Configuration::isSink(DataFlow::Node sink)`
  - `Configuration::hasFlow(DataFlow::Node src, DataFlow::Node sink)`: holds if data may flow from `src` to `sink` for this configuration.

Global Data Flow Analysis: Example

```
import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.FlowSources

class UserControlledPathConfiguration extends DataFlow::Configuration {
  UserControlledPathConfiguration() { this = "UserControlledPathConfiguration" }

  override predicate isSource(DataFlow::Node node) { node instanceof RemoteFlowSource }

  override predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess ma |
      ma.getMethod().hasQualifiedName("java.nio.file", "Path", "of") and
      ma.getAnArgument() = sink.asExpr()
    )
  }
}

from DataFlow::Node source, DataFlow::Node sink, UserControlledPathConfiguration cfg
where cfg.hasFlow(source, sink)
select sink, "The location of this path depends on user-controlled input."
```

Data Flow Analysis vs. Taint Tracking

- Data flow:
  - Value is preserved at each step.
  - In `x = z; y = x + 1;`, data will flow from `z` to `x` but not to `y`.
  - `x + 1` does not preserve the value.
- Taint Tracking
  - Value doesn't have to be preserved at each step.
  - Being influenced or *tainted* is enough.
  - In `x = z; y = x + 1;`, data will flow from `z` to `x` and `x` will taint `y`.
- Taint tracking is an **extension** of data flow and includes steps that not necessarily preserve the data value.

Taint Tracking: Flavors

- Local
  - Follows taint within a single function.
  - Precise and relatively cheap.
  - `TaintTracking::localTaint`, `DataFlow::localExprTaint`
- Global
  - Follows taint across functions.
  - Less precise and computationally expensive.
  - `TaintTracking::Configuration`, `Configuration::hasFlow`

Real World

Apache Commons JXPath

- Apache Commons JXPath (https://github.com/apache/commons-jxpath):

  > A Java-based implementation of XPath 1.0.

- CVE-2022-41852:

  > Those using JXPath to interpret untrusted XPath expressions may be vulnerable to a remote code execution attack. All JXPathContext class functions processing a XPath string are vulnerable except compile() and compilePath() function.

- CVE has been **rejected**, because "Input to JXPath is expected to be administrator controlled and therefore trusted."
- Question: Is input really always trusted? Spoiler: No (if you live in reality)!

Finding CVE-2022-41852 Using CodeQL

- CVE-2022-41852:

  > Those **using JXPath to interpret untrusted XPath expressions** may be vulnerable to a remote code execution attack. **All JXPathContext class functions processing a XPath string** are vulnerable except compile() and compilePath() function.

⇒ perfect fit for taint-tracking analysis.

- Source: Any untrusted input.
- Sink: Any `JXPathContext` function that takes a XPath string except `compile()` and `compilePath`.

Modeling the Source

- **Exercise 7**:
  - Find all nodes that are sources of untrusted data.
  - Hint: `semmle.code.java.dataflow.FlowSources`.

Modeling the Source: Solution

- Solution:
  - `semmle.code.java.dataflow.FlowSources` defines various flow sources for taint tracking.
  - `RemoteFlowSource`: Represents a data flow source of remote user input.

-
  ```
  import java
  import semmle.code.java.dataflow.FlowSources
  import semmle.code.java.dataflow.DataFlow

  predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }
  ```
- **That's everything!**

Modeling the Sink

- **Exercise 8**:
  - > Sink: Any `JXPathContext` function that takes a XPath string except `compile()` and `compilePath`.
  - Find all calls to this sink.
  - Hint: Looking at the source code we know that the following functions are vulnerable: `createPath`, `createPathAndSetValue`, `getPointer`, `getValue`, `iterate`, `iteratePointers`, `removeAll`, `removePath`, `selectNodes`, `selectSingleNode`, `setValue`

Modeling the Sink: Solution

- Solution:
    - The following functions are vulnerable: `createPath` , `getPointer` , `createPathAndSetValue` , `getValue` , `iterate` , `iteratePointers` , `removeAll` , `removePath` , `selectNodes` , `selectSingleNode` , `setValue`

-
  ```
  predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess ma |
      ma.getMethod()
          .hasQualifiedName("org.apache.commons.jxpath", "JXPathContext",
            [
              "createPath", "createPathAndSetValue", "getPointer", "getValue", "iterate",
              "iteratePointers", "removeAll", "removePath", "selectNodes", "selectSingleNode",
              "setValue"
            ]) and
      ma.getArgument(0) = sink.asExpr()
    )
  }
  ```

Putting Everything Together

- **Exercise 9**:
  - Define a taint-tracking configuration with the source and sink we just defined.
  - Hint: Type "taint" in your IDE and hit auto-complete to generate boilerplate for a taint-tracking configuration.
  - Run it!
- You should find **1** result.

Putting Everything Together: Solution

- Solution:
-

```
import java
import semmle.code.java.dataflow.TaintTracking
import semmle.code.java.dataflow.FlowSources

class JXPathInjectionTracking extends TaintTracking::Configuration {
  JXPathInjectionTracking() { this = "JXPathInjectionTracking" }

  override predicate isSource(DataFlow::Node source) { source instanceof RemoteFlowSource }

  override predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess ma |
      ma.getMethod()
          .hasQualifiedName("org.apache.commons.jxpath", "JXPathContext",
            ["createPath", "createPathAndSetValue", "getPointer", "getValue", "iterate",
              "iteratePointers", "removeAll", "removePath", "selectNodes", "selectSingleNode", "setValue"]) and
      ma.getArgument(0) = sink.asExpr()
    )
  }
}

from JXPathInjectionTracking cfg, DataFlow::Node src, DataFlow::Node sink
where cfg.hasFlow(src, sink)
select sink, src, sink, "User-controlled data in XPath expression can lead to RCE."
```

Putting Everything Together: A Better Solution

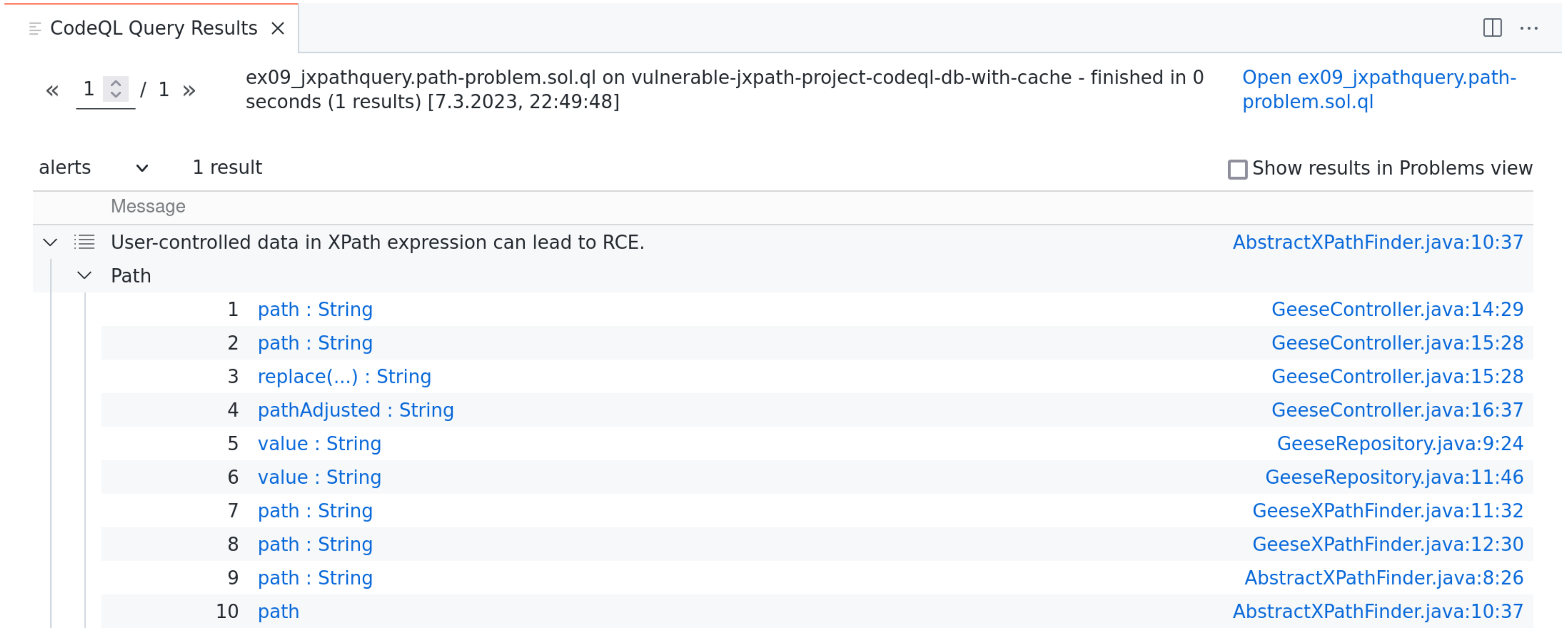
- With the current solution we know that data flows from a source to a sink.
- What we **really** want is to see the *actual* steps the data takes!
- ⇒ we want a *path-problem*.

Putting Everything Together: Path-problem Solution

```
/**
* // CHANGED: Add this, so CodeQL/the VS Code extension knows to interpret the results as a path.
* @kind path-problem
*/

import java
import semmle.code.java.dataflow.TaintTracking
import semmle.code.java.dataflow.FlowSources
// CHANGED: Add this, so data flow queries "generate" results as a path.
import DataFlow::PathGraph

class JXPathInjectionTracking extends TaintTracking::Configuration {
  // unchanged [...]
}

// CHANGED: Instead of `DataFlow::Node`, we have to use `DataFlow::PathNode`.
from JXPathInjectionTracking cfg, DataFlow::PathNode src, DataFlow::PathNode sink
// CHANGED: Instead of `hasFlow`, we have to use `hasFlowPath`.
where cfg.hasFlowPath(src, sink)
select sink, src, sink, "User-controlled data in XPath expression can lead to RCE."
```

Putting Everything Together: Path-problem Solution

- We can follow the flow through the source code by clicking on any of the steps:



CodeQL Query Results

« 1 / 1 »

ex09_jxpathquery.path-problem.sol.ql on vulnerable-jxpath-project-codeql-db-with-cache - finished in 0 seconds (1 results) [7.3.2023, 22:49:48]

Open ex09_jxpathquery.path-problem.sol.ql

alerts 1 result

☐ Show results in Problems view

	Message	
	User-controlled data in XPath expression can lead to RCE.	AbstractXPathFinder.java:10:37
	Path	
1	path : String	GeeseController.java:14:29
2	path : String	GeeseController.java:15:28
3	replace(...) : String	GeeseController.java:15:28
4	pathAdjusted : String	GeeseController.java:16:37
5	value : String	GeeseRepository.java:9:24
6	value : String	GeeseRepository.java:11:46
7	path : String	GeeseXPathFinder.java:11:32
8	path : String	GeeseXPathFinder.java:12:30
9	path : String	AbstractXPathFinder.java:8:26
10	path	AbstractXPathFinder.java:10:37

Further Steps

GitHub Security Lab

> GitHub Security Lab’s mission is to inspire and enable the community to secure the open source software we all depend on.

- What they do:
    - Find vulnerabilities (Google Chrome, Android, Ubuntu, ...)
    - Share research through proof-of-concepts, articles, tutorials, conferences and community events.
    - Scale security research by performing Variants Analysis for open source projects with CodeQL.
    - **Run a bug bounty program for CodeQL queries** 🎉

CodeQL Bug Bounty

- Write a new CodeQL query for an unmodeled vulnerability class.
- **Awards of up to $6,000 can be granted.**
- Use CodeQL query to find and fix vulnerabilities.
- **Awards of up to $7,800 for multiple critical CVEs can be granted.**

More information: https://securitylab.github.com/bounties/

Tips and Tricks

Some useful tips and tricks for writing and debugging CodeQL queries and for using CodeQL at scale.

- GitHub Copilot
- AST Viewer
- GitHub Codesearch
- Multi-repository variant analysis (MRVA)

GitHub Copilot

GitHub Copilot helps you write software faster by using machine learning to generate code suggestions. - It can even help you write CodeQL queries!
It needs little context to get started and it works well as a helpful assistant.

Demo

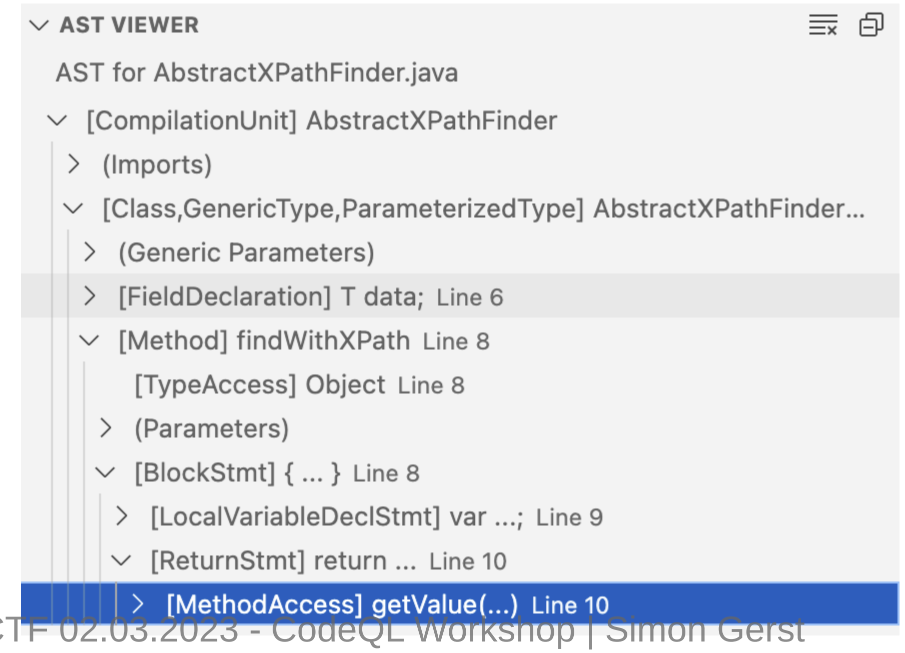
Our context:

```
import java
```

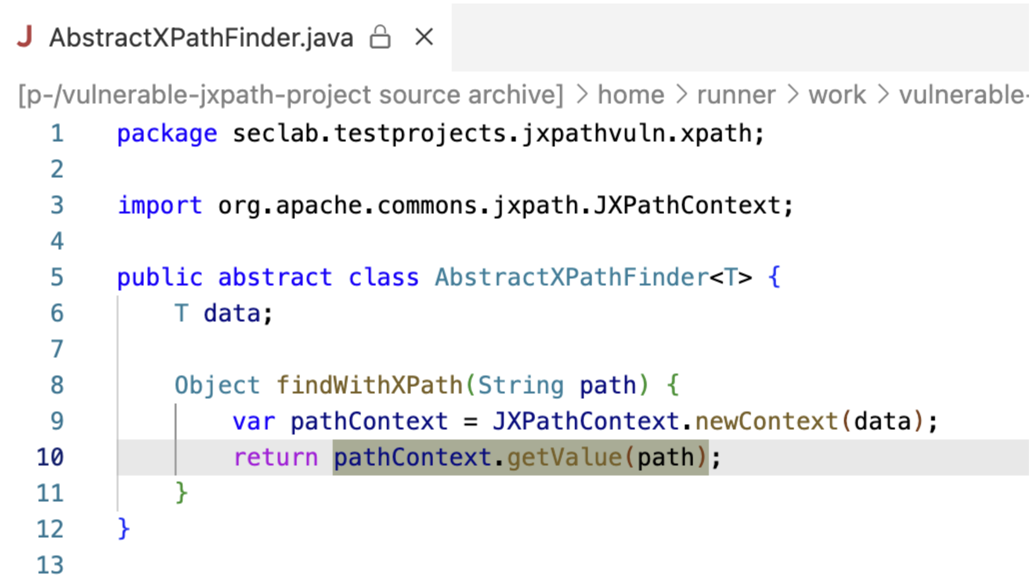
AST Viewer

The AST Viewer is a tool that allows you to view the abstract syntax tree (AST) of a piece of code. It can be used to understand how the CodeQL parser interprets a piece of code.

Demo: Right click "CodeQL: View AST" in result from previous query.



```
AST VIEWER
  AST for AbstractXPathFinder.java
  [CompilationUnit] AbstractXPathFinder
    (Imports)
    [Class,GenericType,ParameterizedType] AbstractXPathFinder...
      (Generic Parameters)
      [FieldDeclaration] T data; Line 6
      [Method] findWithXPath Line 8
        [TypeAccess] Object Line 8
        (Parameters)
        [BlockStmt] { ... } Line 8
          [LocalVariableDeclStmt] var ...; Line 9
          [ReturnStmt] return ... Line 10
            [MethodAccess] getValue(...) Line 10
```



AbstractXPathFinder.java

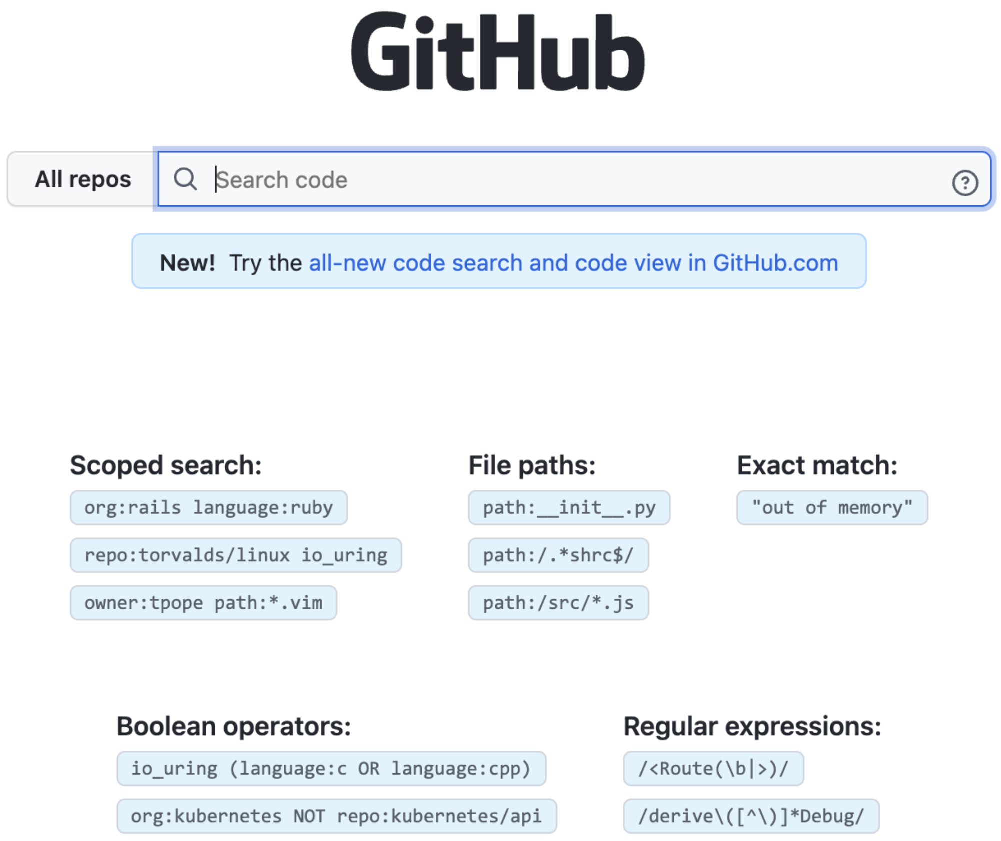
[p-/vulnerable-jxpath-project source archive] > home > runner > work > vulnerable-

```java
package seclab.testprojects.jxpathvuln.xpath;

import org.apache.commons.jxpath.JXPathContext;

public abstract class AbstractXPathFinder<T> {
    T data;

    Object findWithXPath(String path) {
        var pathContext = JXPathContext.newContext(data);
        return pathContext.getValue(path);
    }
}
```

GitHub Codesearch

By enabling the improved GitHub Codesearch at https://cs.github.com you can search for code across all of GitHub. It allows much more powerful queries than the previously existing search functionality on GitHub (you can even use Regex).

The Codesearch startpage



New! Try the all-new code search and code view in GitHub.com

Scoped search:

```
org:rails language:ruby
repo:torvalds/linux io_uring
owner:tpope path:*.vim
```

File paths:

```
path:__init__.py
path:/.*shrc$/
path:/src/*.js
```

Exact match:

```
"out of memory"
```

Boolean operators:

```
io_uring (language:c OR language:cpp)
org:kubernetes NOT repo:kubernetes/api
```

Regular expressions:

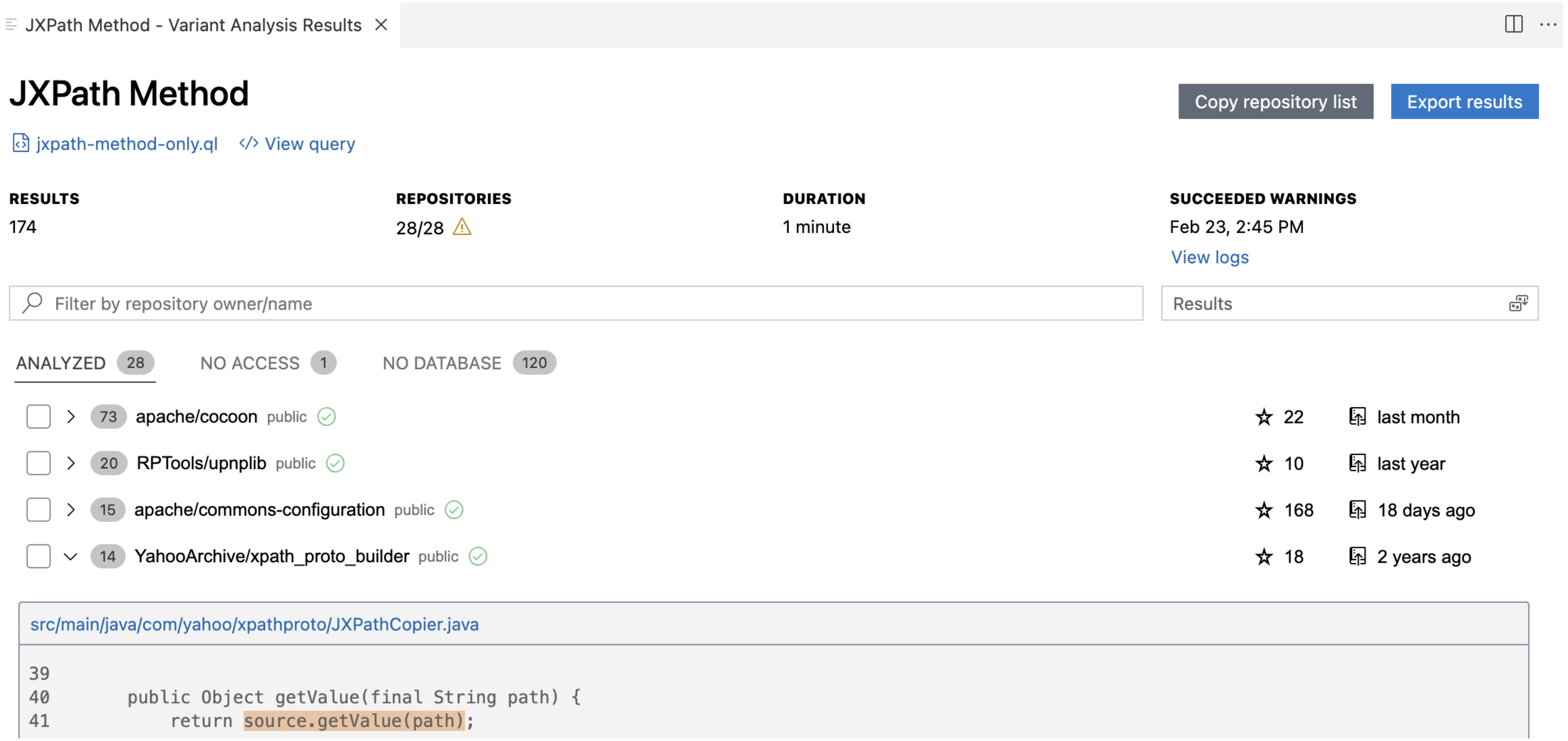
```
/<Route(\b|>)/
/derive\([^\)]*Debug/
```

Multi-repository variant analysis

Multi-repository variant analysis (MRVA) is a feature that allows you to run CodeQL queries across multiple repositories. E.g. you can use Codesearch to identify specific sinks in your organization or OSS projects and then you can leverage MRVA to find all the places where remote data flows into these sources.

MRVA is publicly available for testing since yesterday!

MRVA results inside of VS Code



JXPath Method - Variant Analysis Results ×

JXPath Method

RESULTS	REPOSITORIES	DURATION	SUCCEEDED WARNINGS
174	28/28 ⚠	1 minute	Feb 23, 2:45 PM

73	apache/cocoon public ✓	☆ 22	last month
20	RPTools/upnplib public ✓	☆ 10	last year
15	apache/commons-configuration public ✓	☆ 168	18 days ago
14	YahooArchive/xpath_proto_builder public ✓	☆ 18	2 years ago

src/main/java/com/yahoo/xpathproto/JXPathCopier.java

```

    public Object getValue(final String path) {
        return source.getValue(path);
```

Summary

- Thank YOU for attending today and we hope you learned something.
- You now know the basics (and more) of CodeQL and modeled a real-world CVE.
- Your skills are transferable to the other languages supported because what changes are only a few keywords/concepts.

Resources

- QL tutorials
- CodeQL language guides
- Other CodeQL workshops
- GitHub Advanced Security material
- QL language reference
- Overview over all CodeQL Java classes

Questions? Concerns? Comments?

Ping `@intrigus` in the GitHub Security Lab Slack.

Join the GitHub Security Lab Slack!

- For all questions regarding:
    - CodeQL
    - Bounties
    - Multi-repository variant analysis
- Join the GitHub Security Lab Slack by following this link: gh.io/securitylabslack.

Backup

Creating a Database

- First, verify building your code using your usual build system: make/cmake/gradle/go/...
- Everything works?
- Compile your code again using the CodeQL cli tools.
- The cli intercepts the build command and extracts the necessary facts from your code while compiling.
- Cli must intercept the actual compile command.
  ⇒ can not use incremental builds or caching.
  (Only code that has *been compiled in a* `codeql` *command* will be included into the database)

Example: Custom Compile Command

- Toy project with a single file.
- Normal command: `javac Example.java`
- Building database command: `codeql database create the_database_name --command 'javac Example.java' --language Java`

Example: Analyzing Multiple Projects Together

- You have a project A and a dependency project B and have created *separate* databases for them.
- When analyzing project A, CodeQL treats calls to methods in B as a black box!
  ⇒ this is a big problem for data flow analysis.
- Solution: Just build both projects in the same `codeql` command.

Example: Analyzing Multiple Projects Together (continued)

- Before: `codeql database create projectA --command '[buildCommandA]' --language Java`
  and `codeql database create projectB --command '[buildCommandB]' --language Java`
- After: `codeql database create projectAB --command '[buildCommandA] && [buildCommandB]' --language Java`
  ⇒ the database `projectAB` now also contains the code of B and so data flow analysis doesn't stop at function boundaries 🎉

Example: Some `gradle` Project Working Out of the Box

- No need to provide an explicit build command.
- CodeQL provides an "autobuilder":
  detects the build system and executes an appropriate command.
- Instead of
  `codeql database create projectA --command 'mvn compile' --language Java`
  you could also just run
  `codeql database create projectA --language Java`

Example: Other Languages

- Python, Javascript/Typescript, and Ruby:

  No need for a compilation command.

  ```
  $ codeql database create $DB_NAME --language {Python|Javascript|Ruby}
  ```
- C#:

  ```
  $ codeql database create $DB_NAME --language csharp --command
  '[buildCommand]
  ```
- Go:

  ```
  $ codeql database create $DB_NAME --language go --command
  '[buildCommand]
  ```

Predicates with Result

- Predicates can also have a result: replace `predicate` with the result type.
- "Return" a value by expressing a relation between `result` and the other variables.
- Example:

```
int getSuccessor(int i) {
  result = i + 1 and
  i in [1 .. 9]
}
```

Predicates: Examples

```
// (BlockStmt): is `block` an empty block?
predicate isEmpty(BlockStmt block) {
  block.getNumStmt() = 0
}

// (Expr, Call): get the first argument of a function call.
Expr getFirstArg(Call call) {
  result = call.getArgument(0)
}

// (Expr, Call): same as before but expressed differently.
predicate getFirstArg(Expr expr, Call call) {
  expr = call.getArgument(0)
}
```

Predicates != Functions

```
// (Call, Expr): Get all Call’s that have `expr` as its first argument.
Call getCallFor(Expr expr) {
  expr = result.getArgument(0)
}

// (Call, Expr): Get all Calls that have `expr` as its first argument.
predicate getCallFor(Call call, Expr firstArg) {
  firstArg = call.getArgument(0)
}

// (int, Call, Expr): Get the index of `arg` when called as an argument
// of `call`.
int getArgIndex(Call call, Expr arg) {
  arg = call.getArgument(result)
}
```

Transitive Closures

- Apply the same predicate zero or more times in a transitive way.
- It allows you to express relationships between elements of the same set that are "connected" by means of a relationship/predicate.

Transitive Closures: Example

- `Class class.getASupertype()`: Gets a super type of `class`. Elements are related by whether they are super types of each other.
- `c.getASupertype*().hasQualifiedName("java.io", "Reader")`: Gets all classes that have `java.io.Reader` as direct super type.
- `c.getASupertype+().hasQualifiedName("java.io", "Reader")`: Gets all classes that have `java.io.Reader` as direct or indirect super type (**ex**cluding `Reader` itself).
- `c.getASupertype*().hasQualifiedName("java.io", "Reader")`: Gets all classes that have `java.io.Reader` as direct or indirect super type (**in**cluding `Reader` itself).

Equality

- Equality can be a bit surprising:
  - There is a difference between `A != B` and `not A = B`!
- `A = B` holds if there is a pair of values — one from `A` and one from `B` — that are the same. In other words, A and B have at least one value in common.
- `A != B` holds if there is a pair of values (one from `A` and one from `B`) that are different.
- `not A = B` holds if it is not the case that there is a pair of values that are the same. In other words, `A` and `B` have no values in common.

More information: https://codeql.github.com/docs/ql-language-reference/formulas/#equality

Equality: Example

- No difference between `A != B` and `not A = B` when `A` and `B` contain a single value
  - `0 = 1` does not hold, `0 != 1` holds and `not 0 = 1` holds.
- Now we compare `1` and `[1..2]`
  - `1 != [1..2]` holds, because `1 != 2`.
  - `1 = [1..2]` holds, because `1 = 1`.
  - `not 1 = [1..2]` doesn’t hold, because there is a common value (`1`).

Data Flow/Taint Tracking: Limitations

- Flow/taint can not propagate through external functions.

-
  ```
  String result someExternalFunction(taintedStringValue);
  ```
- Is `result` tainted? 🤷 **impossible** to know without source code!
- Many important external functions are from the Java runtime.
  ⇒ have to define manual models
- "If parameter `foo` is tainted, then the result of the call to `bar` should also be considered tainted."

Query Metadata

Add header comment `/** ... */` at beginning of QL file to provide metadata:

- `@name`: a descriptive name of the query
- `@description`: a short description of what the query does
- `@id`: a unique identifier for the query
- `@kind`: how to display results, e.g. `@kind problem` for alert queries

More information: https://codeql.github.com/docs/writing-codeql-queries/metadata-for-codeql-queries/

Path Problems

- Knowing that data flows from a source to a sink is good.
- But seeing the actual flow path is better.
- Path queries require certain metadata, query predicates, and select statement structures.

-
  ```
  /**
   * @kind path-problem
   */
  import DataFlow::PathGraph
  ...

  from Configuration config, DataFlow::PathNode source, DataFlow::PathNode sink
  where config.hasFlowPath(source, sink)
  select <alert location>, source, sink, "<message>"
  ```

Path Problems: Example

```
/**
 * @kind path-problem
 */

import java
import semmle.code.java.dataflow.DataFlow
import DataFlow::PathGraph
...

from JXPathInjectionTracking config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "User-controlled data in XPath expression can lead to RCE."
```