GitHub 2021

Introduction to CodeQL

Java

Presented by @s-samadi, @xaverixmorris, and @hohn

Meet the Team

Shadi Samadi - Security Solution Architect

Xavier Morris - CodeQL Analysis Engineer

Michael Vesic- Project Manager

Michael Hohn - CodeQL Analysis Engineer

Today we will cover

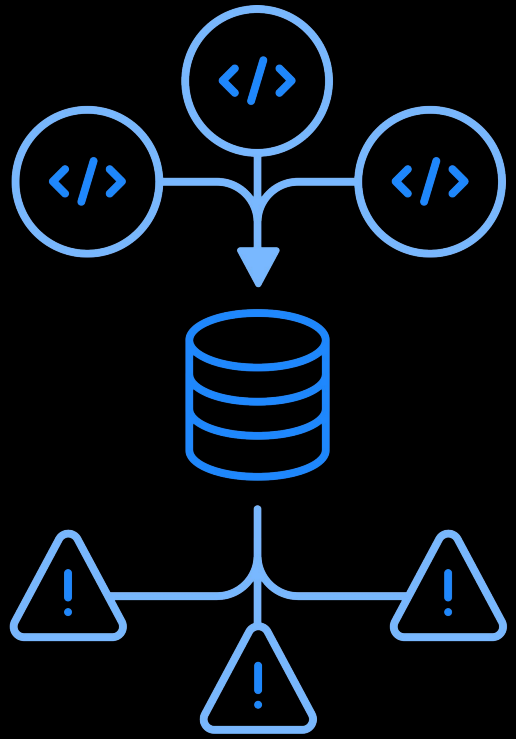
- Overview of CodeQL
- How CodeQL Works
- CodeQL Language and Concepts
- SQL Injection Vulnerabilities
- Exercise

CodeQL

Analyze code as data using expressive queries to say what you want to find, not how to find it

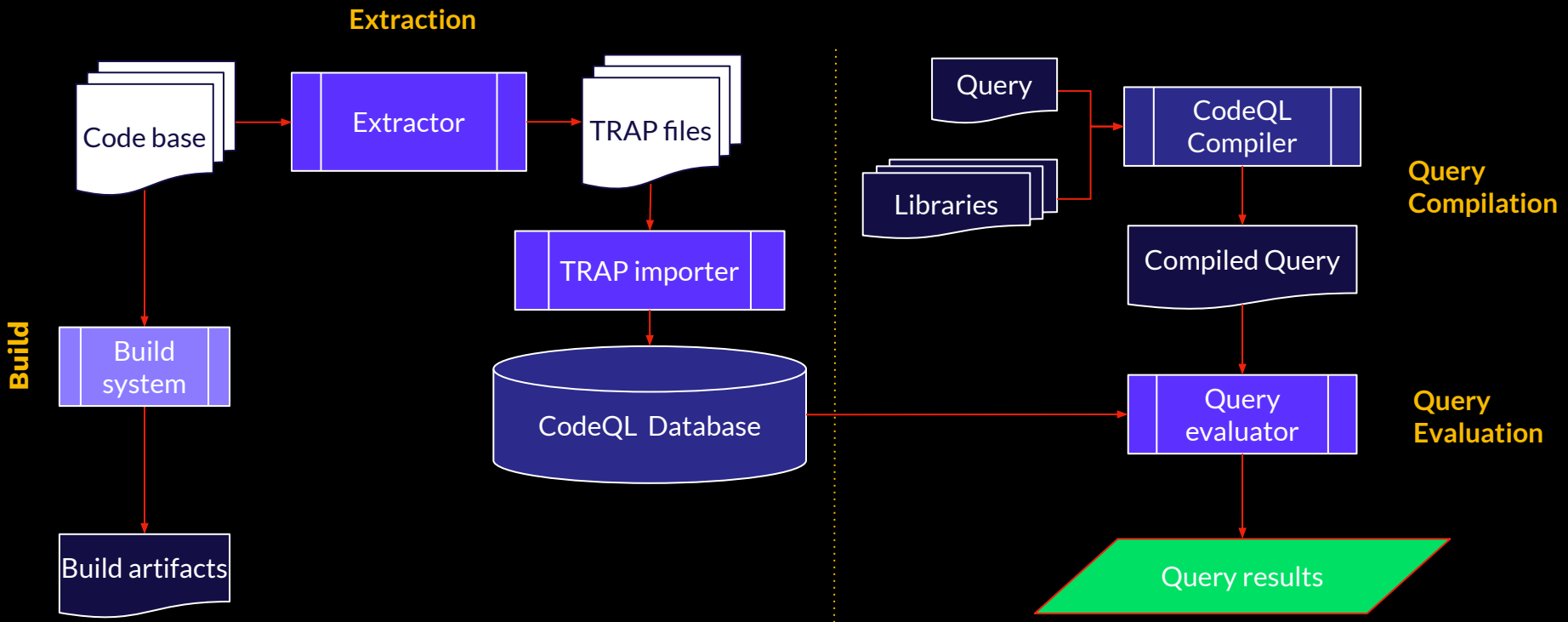
Quickly refine analyses to increase precision within your codebase

Share security knowledge within your teams using codified, readable and executable queries



Introduction to CodeQL

How CodeQL works



CodeQL Language

The basic syntax will look familiar to anyone who has used SQL, but it is used somewhat differently.

- CodeQL is a logic programming language. CodeQL uses common logical connectives (such as `and`, `or`, and `not`).
- Quantifiers (such as `forall` and `exists`)
- CodeQL supports recursion and aggregates.
- CodeQL is object-oriented - supports classes so you can define new high level entities to represents or extend a given model.

Simple query that return `"hello world"`.

```
import <language>

select "hello world"
```

More complicated queries typically look like this:

```
from /* ... variable declarations ... */
where /* ... logical formulas ... */
select /* ... expressions ... */
```

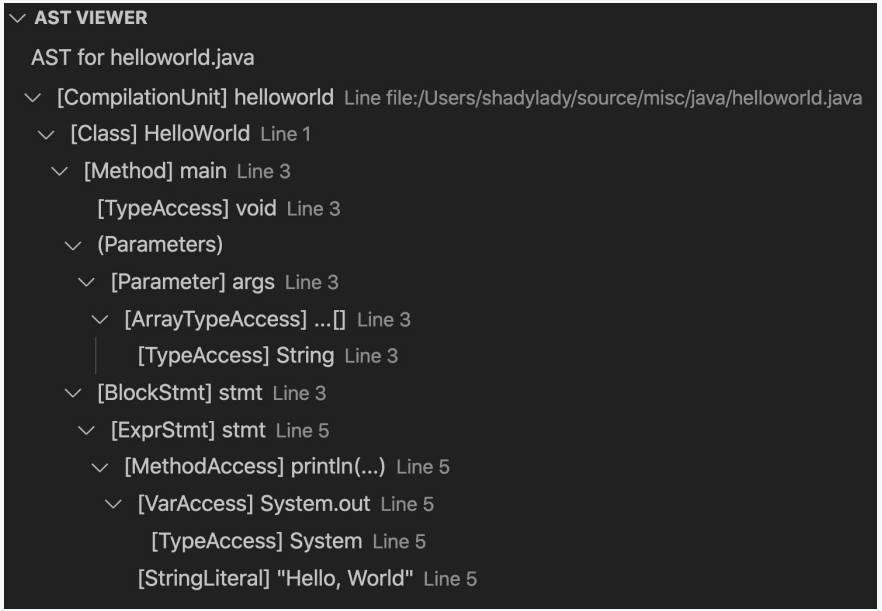
For example, the result of this query is the number 42:

```
from int x, int y
where x = 6 and y = 7
select x * y
```

An example of a Class declaration

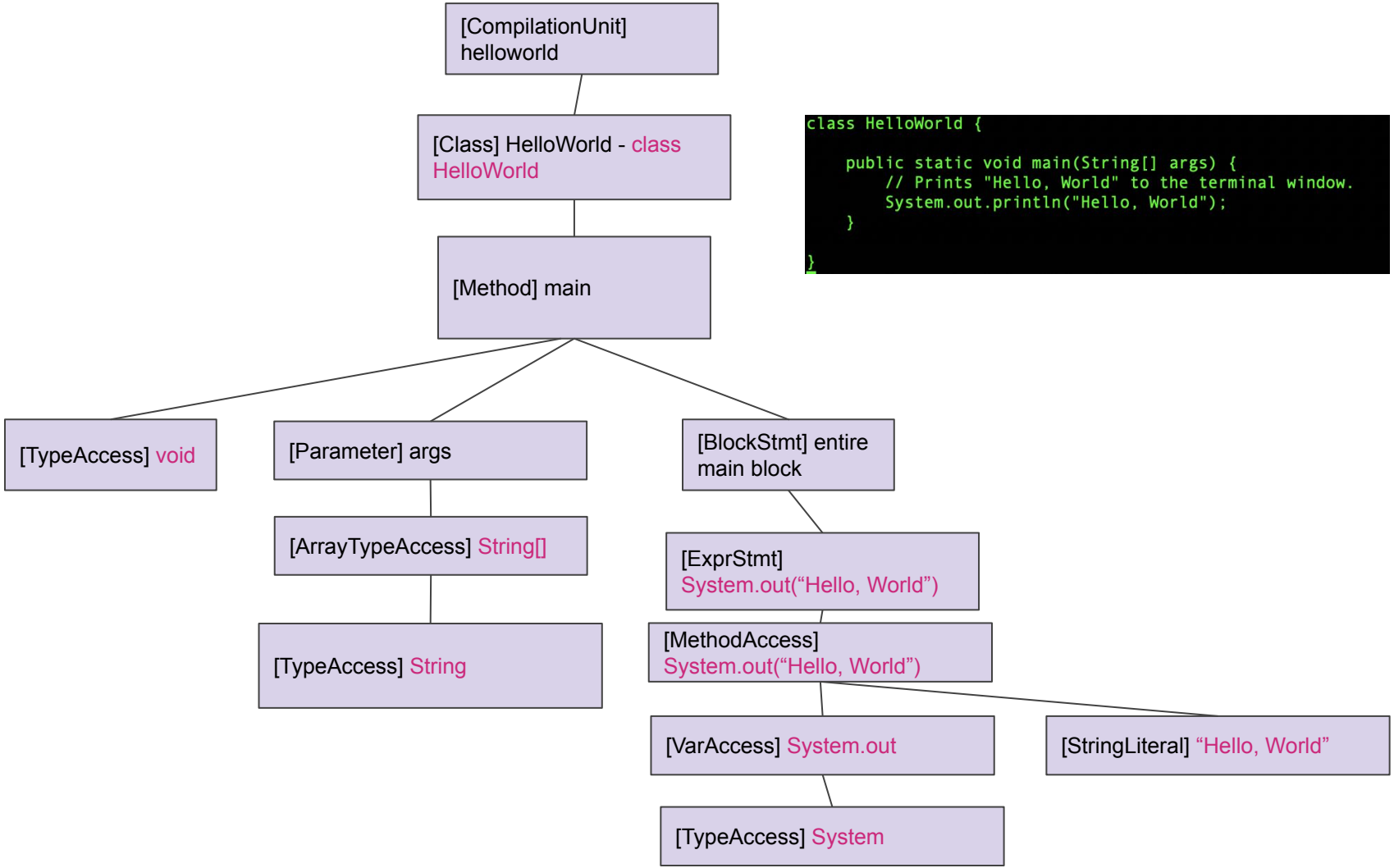
```
class SmallInt extends int {
  SmallInt() { this in [1..10] }
  int square() { result = this*this }
}

from SmallInt x, SmallInt y, SmallInt z
where x.square() + y.square() = z.square()
select x, y, z
```

Abstract Syntax Tree - AST



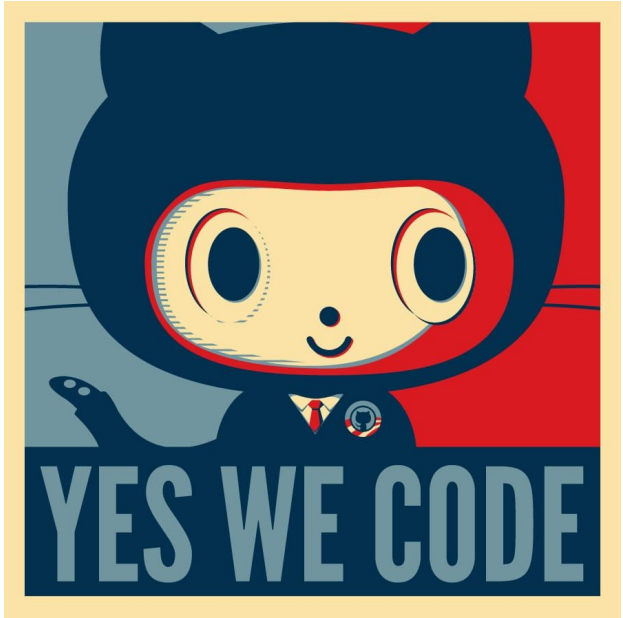
- Generated by the extractor
- The abstract syntax tree (AST) of a program represents the program's syntactic structure
- AST Viewer in VSCode
- The integrated viewer show MOST of the AST, but not every detail

```
class HelloWorld {

    public static void main(String[] args) {
        // Prints "Hello, World" to the terminal window.
        System.out.println("Hello, World");
    }

}
```


[CompilationUnit] helloworld
[Class] HelloWorld - class HelloWorld
[Method] main
[TypeAccess] void
[Parameter] args
[BlockStmt] entire main block
[ArrayTypeAccess] String[]
[TypeAccess] String
[ExprStmt] System.out(“Hello, World”)
[MethodAccess] System.out(“Hello, World”)
[VarAccess] System.out
[StringLiteral] “Hello, World”
[TypeAccess] System
class HelloWorld {
public static void main(String[] args) {
// Prints "Hello, World" to the terminal window.
System.out.println("Hello, World");
}
}

Control Flow Graph

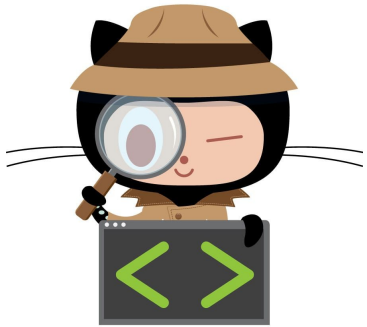
- Control flow creates a graph from AST
- Models the order of evaluation
- Typically used to determine if something is evaluated before or after another AST node - predecessor or successor
- Modelling domination path - control flow X dominates control flow Y if all the control flow paths to Y have to go through X first



Dataflow Analysis

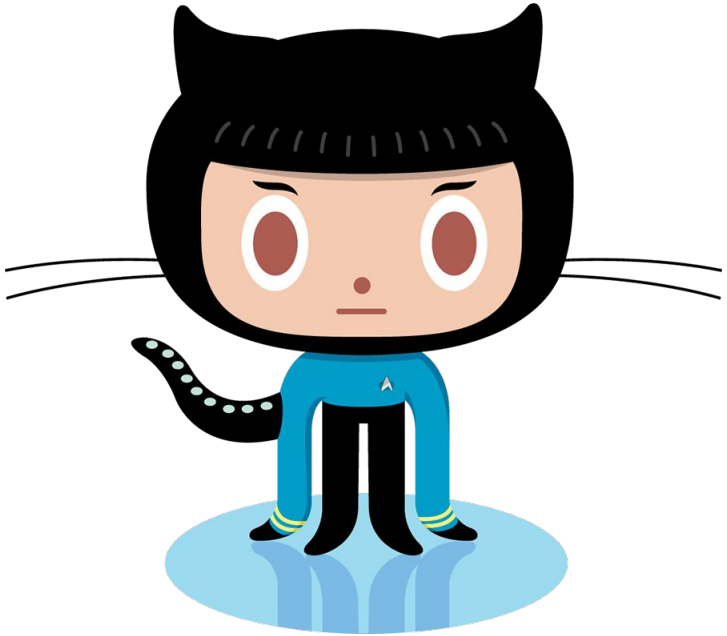
We model the flow of data through the program as a directed graph, where:

- The *nodes* of the graph represent program elements that have a value, such as expressions and parameters.
- The *edges* of the graph represent *flow* between those program elements.



Local vs Global data flow

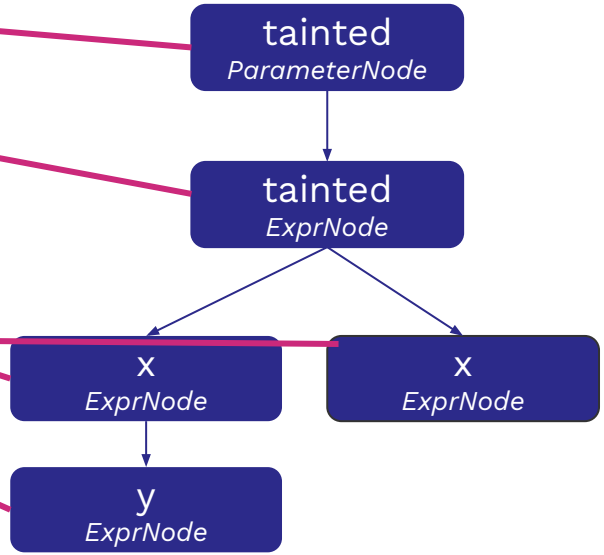
- Local (“intra-procedural”) data flow models flow within one function; feasible to compute for all functions in a snapshot
- Global (“inter-procedural”) data flow models flow across function calls; not feasible to compute for all functions in a snapshot



Dataflow graphs

```
public void(int tainted) {
  int x = tainted;
  if(somecondition) {
     int y = x;
     callFoo(y);
   } else {
     return x;
   }
   return - 1;
}
```

Data flow graph:



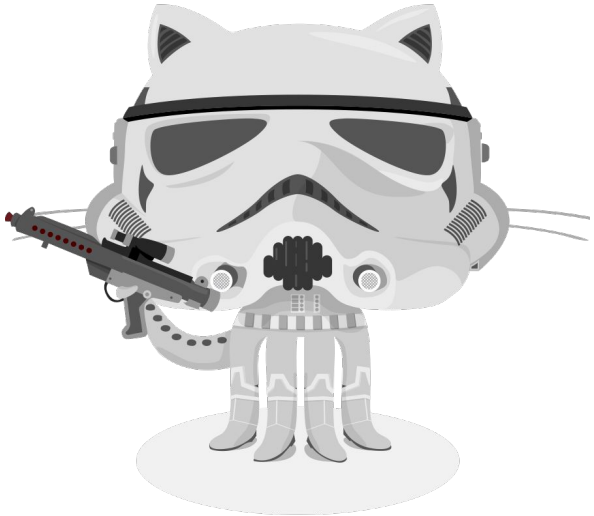
Taint Tracking

```
public void foo() {
  string formatString = "hello";
  string tainted = "world!";

  formatString = formatString + tainted;

}
```

- Data flow analysis tells us how values flow *unchanged* through the program. Taint tracking analysis is slightly more general: it tells us how values flow through the program and may undergo minor changes, while still influencing (or tainting) the places where they end up.
- Would not be found by data flow alone because concatenation modifies the `formatString` by *appending* the tainted value.
- Taint tracking can be thought of as another type of data flow graph. It usually extends the standard data flow graph for a problem by adding edges between nodes where one one node influences or taints another.

The node types

- AST Nodes
- Data flow Nodes
- Control flow Nodes



Exercise

SQL Injection

- Application accepts user input data that results in the injection of SQL commands that execute on the database
- OWASP Security Shepherd - Java repo with deliberate vulnerabilities for training and educational purposes
- https://github.com/OWASP/SecurityShepherd

SQL Injection - Problem Statement

- Source

```
String CheckName = username.getText().toString();
String CheckPass = password.getText().toString();
```

- Sink

```
Cursor cursor = db.rawQuery(query, null);
```

- Dataflow between source and sink