Nullcon Berlin 2024 - CodeQL Workshop

Setup instructions: https://gh.io/nc-2024-setup

Your CodeQL Guide Today

Simon Gerst

CS Student & GitHub Security Ambassador

github.com/intrigus-lgtm

twitter.com/intrigus_

infosec.exchange/@intrigus

`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 me and ask!

Checkpoint: Setup

- Did you follow the setup instructions on https://gh.io/nc-2024-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:

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 cpp`: imports the C/C++ query library; makes classes and predicates available
- `from StringLiteral sl`: from **all** elements that represent string literals
- `where sl.getValue() = "Hello, World!\n"`: only get the strings that represents "Hello, World!\n".
- `select sl, "Hello, World!\\n found"`: create an alert.

Examples

```
select "Hello Nullcon"
```

Examples

```
from int year
where year = 2024
select "Hello Nullcon " + year
```

Examples

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

Examples

```
from StringLiteral sl
where sl.getValue() = "Hello, World!\n"
select sl, "Hello, World!\\n found"
```

Building Blocks

Building Blocks for C/C++

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

Declarations

- Represent C/C++ declarations:
- Types (`Type`).
- Functions (`Function`).
- Constructors (`Constructor`).
- Variables (`Variable`).
- Helpful member predicates:
  - `hasGlobalOrStdName(string name)`: Holds if this declaration has the given name in the global namespace or the std namespace.

Declarations: Variables

- `Variable` represents variables in the C/C++ sense:
    - `GlobalVariable` represents a global variable.
    - `MemberVariable` represents any member variable.
    - `Field` represents a non-static member variable.
    - `LocalVariable` represents a local variable.
    - `Parameter` represents a parameter of a function or constructor.
- Helpful member predicates:
    - `VariableAccess Variable.getAnAccess()`: Gets an access to this variable.
    - `Expr Variable.getAnAssignedValue()`: Gets an expression that is assigned to this variable.

Declarations: Types

- Class `Type` represents different kinds of C/C++ types:
  - `BuiltInType` represents a built-in primitive type: `int`, `float`, `void`, ...
  - `UserType` represents all user-defined type; it has several subclasses:
    - `Class` represents a C/C++ `struct`, `union`, `class` type.
    - `Enum` represents a C/C++ `enum` type.
  - `DerivedType` represents a C/C++ type formed from another type.
    - `Array` represents a C/C++ array type.
    - `PointerType` represents a C/C++ pointer type.

Declarations: Types (Continued)

- Helpful member predicates:
  - `Class.getAMemberFunction`: Gets a function declared in this type.
  - `Class.getAMemberVariable`: Gets a variable declared in this type.
  - `Class.getABaseClass`: Gets a direct supertype of this type.
  - `Class.hasQualifiedName(string namespace, string type, string baseName)`: Holds if this type is in `namepace` and declared in `type` with `baseName`.
  - Example: `std::vector::size()`: `hasQualifiedName("std", "vector", "size")`

Declarations: Functions

- `Function`: Represents C/C++ functions, for example, `void *malloc(size_t size)`
- Helpful member predicates:
  - `Function.hasName(string name)`: holds if this function has the specified name.
  - `Function.getACallToThisFunction`: get a **call** to this function.
  - `Function.getAnAccess`: get an **access** to this function.
  - `MemberFunction.getAnOverridingFunction`: get a directly overriding function.

Declarations: FunctionCall vs. FunctionAccess

```
void foo() {}
void bar() {
  foo(); // <-- function call
  void (*baz)() = &foo; // <-- function access
}
```

- `FunctionCall`: Represents a function call with a list of arguments; example: `foo()`
  - Target function known at **compile** time.
- `FunctionAccess`: Represents an access to a function; example: `&foo`
  - Target function known at **run** time.
- Mutually exclusive: a function call is not a function access and vice versa.

Declarations: Constructors

- `Constructor`: Represents C++ constructors, for example, `Bar(int) {}`
- `ConstructorCall`: Represents constructor calls.
- ```
  Bar b = Bar(314); // <-- constructor call
  Bar b2; // <-- constructor call
  ```
- ```
  Bar::Bar() : Bar(42) { // <-- `Bar(42)` is constructor call
  }

  Bar::Bar(int x) {}
  ```

Declarations: Calls and Functions

- `Function`: Common super class for `TopLevelFunction` and `MemberFunction`.
  - `MemberFunction`: Common super class for `Constructor`, `VirtualFunction`, ...
- `Call`: Common super class for `FunctionCall` and `ExprCall`.

Exercises: Declarations

Exercise 1: Find all variables named "basedir" of type `std::string`.

Hints:

- Get the type of a variable using the `getType()` method.
- Use `hasName(name)` to check whether a type has a specific name.

Exercise 2: Find all functions that have a parameter of type `std::string` and are in a sub namespace of "intrigus::testprojects"

Hint: Match the start of a string like this: `matches("intrigus::testprojects::%")`

(Optional) Exercise 3: Find all member variables whose name ends with "json_string".

Abstract Syntax Tree Nodes

- Abstract syntax trees (ASTs) represent the structure of program code.
- C/C++'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/codeql-library-for-cpp/#statement-classes

Abstract Syntax Tree Nodes (Continued)

- Helpful member predicates:
  - `Expr.getAChild` and `Stmt.getAChild` return a child of a given expression/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 Stmt`	IfStmt
`while ( Expr ) Stmt`	WhileStmt
`return Expr ;`	ReturnStmt

Abstract Syntax Tree Nodes: Expressions

Expression syntax	CodeQL class
Literals: `'c'`, `0xffff`, `"Hello"`, ...	CharLiteral, HexLiteral, StringLiteral, ...
Unary expressions: `Expr++`, `--Expr`, `!Expr`, ...	PostfixIncrExpr, PrefixDecrExpr, NotExpr, ...
Binary expressions: `Expr * Expr`, `Expr && Expr`, `Expr < Expr`, ...	MulExpr, LogicalAndExpr, 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()`, ...	VariableAccess, FieldAccess, ArrayExpr, FunctionCall, ...

Exercises: Abstract Syntax Tree Nodes

Exercise 4: Find all calls to functions with name `PrintValue` that **do not** take place inside functions with names starting with `Dump`.
Hint: Use auto-completion on `FunctionCall` to find the predicate that checks whether a call is enclosed in a function.

Exercise 5: Find all calls to `system` that **do not** use a constant string as the first argument.
Hints:

- Use `StringLiteral` to check for instances 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 functions named `system`.
- Calls to functions are represented by `FunctionCall`.
  ⇒ super type is `FunctionCall`
- "Take all calls to functions, but only those named `system`. Give those values the name `SystemFunctionCall`"

```
class SystemFunctionCall extends FunctionCall {
  SystemFunctionCall() { // <- characteristic predicate
    this.getTarget().hasName("system")
  }
}
```

Explicit Type Checks

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

```
from FunctionCall fc
where fc instanceof SystemFunctionCall
select fc, "call to system"
```

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: C/C++ vs. CodeQL

- Casts in C/C++ **can** throw an exception/result in UB.
- Casts in CodeQL **do not** throw an exception.

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

```
predicate foo(FunctionCall fc) {
  fc.(SystemFunctionCall).cmdArgument().
  (FunctionCall).getTarget().(MemberFunction).hasName("c_str")
}
```

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-cpp-new/

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.
  - Define `DataFlow::ConfigSig` + `DataFlow::Global<MyFlowConfiguration>`

AST Nodes vs. Data Flow Nodes

- AST nodes reflect the **syntactic structure** of the program.
- Classes: `Expr`, `FunctionCall`, `VariableAccess`, ...
- 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`

AST Nodes vs. Data Flow Nodes (Continued)

- We often translate between AST and data flow nodes:

  `Expr DataFlow::Node.asExpr()`: Gets the expression corresponding to this node, if any.

  `Parameter DataFlow::Node.asParameter()`: Gets the positional parameter corresponding to this node, if any.

- ```
  int main(int argc, char *argv[]) {
    ...
  }
  ```

- ```
  predicate isSource(DataFlow::Node source) {
      exists(Parameter p | p.hasName("argv") | p = source.asParameter())
  }
  ```

Local Data Flow Analysis

- Follows propagation within a single function.
- Import `semmle.code.cpp.dataflow.new.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 `system(argv[1])`, but also found constructs like this:

```
char *cmd = "id";
system(cmd);
```

We also want to ignore string literals that are indirect arguments to `system`.

Hints:

- `import semmle.code.cpp.dataflow.new.DataFlow`
- Use `DataFlow::localExprFlow` to track the flow of string literals to `system`.

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.cpp.dataflow.new.DataFlow` and implement `DataFlow::ConfigSig` for module `Foo`.
  - Implement `ConfigSig::isSource(DataFlow::Node src)`
  - Implement `ConfigSig::isSink(DataFlow::Node sink)`
  - `Foo::flow(DataFlow::Node src, DataFlow::Node sink)`: holds if data may flow from `src` to `sink` for `Foo` config.

Global Data Flow Analysis: Pointers and Indirections

```
int main(int argc, char *argv[]) {
  system(argv[1]);
}
```

- Assume that program arguments are user-controlled.
- What exactly is user-controlled?
  - `argv` itself?
  - `argv[1]`?
  - `*argv[1]`?

Global Data Flow Analysis: Pointers and Indirections (Continued)

- Only after dereferencing `argv` twice do we get to the actual user-controlled data.
- ```
  predicate isSource(DataFlow::Node source) {
      exists(Parameter p | p.hasName("argv") | p = source.asParameter())
  }
  ```
- Wrong! `argv` itself is not user-controlled.
  - We have to dereference `argv` 2 times to get to the actual user-controlled data.
- ```
  predicate isSource(DataFlow::Node source) {
      exists(Parameter p | p.hasName("argv") | p = source.asParameter(2))
  }
  ```

Global Data Flow Analysis: Pointers and Indirections (Continued 2)

- Only after dereferencing `cmd` once in `system` do we get to the actual user-controlled data.
- ```
  predicate isSink(DataFlow::Node sink) {
      exists(SystemFunctionCall sfc | sfc.cmdArgument() = sink.asExpr())
  }
  ```
- Wrong! `cmd` itself is not user-controlled.
  - We have to dereference `cmd` 1 time to get to the actual user-controlled data.
- ```
  predicate isSink(DataFlow::Node sink) {
      exists(SystemFunctionCall sfc | sfc.cmdArgument() = sink.asIndirectExpr(1))
  }
  ```

Global Data Flow Analysis: Example

```
import cpp
import semmle.code.cpp.dataflow.new.TaintTracking

module ArgvToSystemConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    exists(Parameter p | p.hasName("argv") | p = source.asParameter(2))
  }

  predicate isSink(DataFlow::Node sink) {
    exists(SystemFunctionCall sfc | sfc.cmdArgument() = sink.asIndirectExpr(1))
  }
}

module ArgvToSystemFlow = TaintTracking::Global<ArgvToSystemConfig>;

from DataFlow::Node source, DataFlow::Node sink
where ArgvToSystemFlow::flow(source, sink)
select sink, "Data flow from argv to system"
```

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`, `TaintTracking::localExprTaint`
- Global
  - Follows taint across functions.
  - Less precise and computationally expensive.
  - Define `DataFlow::ConfigSig` + `TaintTracking::Global<MyFlowConfiguration>`

Real World

Executing Other Programs on a Computer

- Which functions can you use to execute another program on a (linux-based) computer?
- `system`
- `exec` family.
- `popen`
- `posix_spawn` family.
- There is another obscure way to execute another program on a computer:
    - `wordexp`

`wordexp`

- `wordexp` is a function that performs word expansion.
- Word expansion is the process of splitting a string into words.
- `foo bar ~` ⇒ `[foo, bar, /home/user]`
- Also does:
  - Expands `~` to the home directory.
  - Substitutes (environment) variables.
  - **Performs command substitution.**
  - ⇒ arbitrary code execution.

CVE-2022-3008 in TinyGLTF

- TinyGLTF:
- > Header only C++11 tiny glTF 2.0 library
- CVE-2022-3008:
- > The tinygltf library uses the C library function `wordexp()` to perform file path expansion on **untrusted paths** that are provided from the input file. This function allows for command injection by using backticks.
  - ⇒ perfect fit for taint-tracking analysis.
  - Source: Any untrusted input.
  - Sink: Any call to `wordexp()`.

Modeling the Source

- **Exercise 7**:
  - Find all nodes that are sources of untrusted data.
  - Hint: `import semmle.code.cpp.security.FlowSources as FS`.
    - `FS` is a module. To check what it contains, type `FS::` and use auto-completion.

Modeling the Source: Solution

- Solution:
  - `semmle.code.cpp.security.FlowSources` defines various flow sources for taint tracking.
  - `FlowSource`: Represents a source of local or remote user input.
- ```
  import cpp
  import semmle.code.cpp.security.FlowSources as FS
  import semmle.code.cpp.dataflow.new.DataFlow

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

Modeling the Sink

- **Exercise 8**:
  - > Sink: All calls to first argument of `int wordexp(char *s, wordexp_t *p, int flags)`.
  - Find all calls to this sink.

Modeling the Sink: Solution

- Solution:
  - `FunctionCall`: Represents a call to a function.
  - `sink.asIndirectExpr(1)`: Taint is found after dereferencing the expressions `1` time.
- ```
  predicate isSink(DataFlow::Node sink) {
    exists(FunctionCall fc |
      fc.getTarget().hasGlobalOrStdName("wordexp") and
      sink.asIndirectExpr(1) = fc.getArgument(0)
    )
  }
  ```
  ```

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 cpp
  import semmle.code.cpp.dataflow.new.TaintTracking
  import semmle.code.cpp.security.FlowSources as FS

  module RemoteToWordexp implements DataFlow::ConfigSig {
    predicate isSource(DataFlow::Node source) { source instanceof FS::FlowSource }

    predicate isSink(DataFlow::Node sink) {
      exists(FunctionCall fc |
        fc.getTarget().hasGlobalOrStdName("wordexp") and sink.asIndirectExpr(1) = fc.getArgument(0)
      )
    }
  }

  module RemoteToWordexpFlow = TaintTracking::Global<RemoteToWordexp>;

  from DataFlow::Node source, DataFlow::Node sink
  where RemoteToWordexpFlow::flow(source, sink)
  select source, "This value flows from a remote source to a 'wordexp' call that executes commands."
  ```
  ```

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!
- $\Rightarrow$ 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
*/

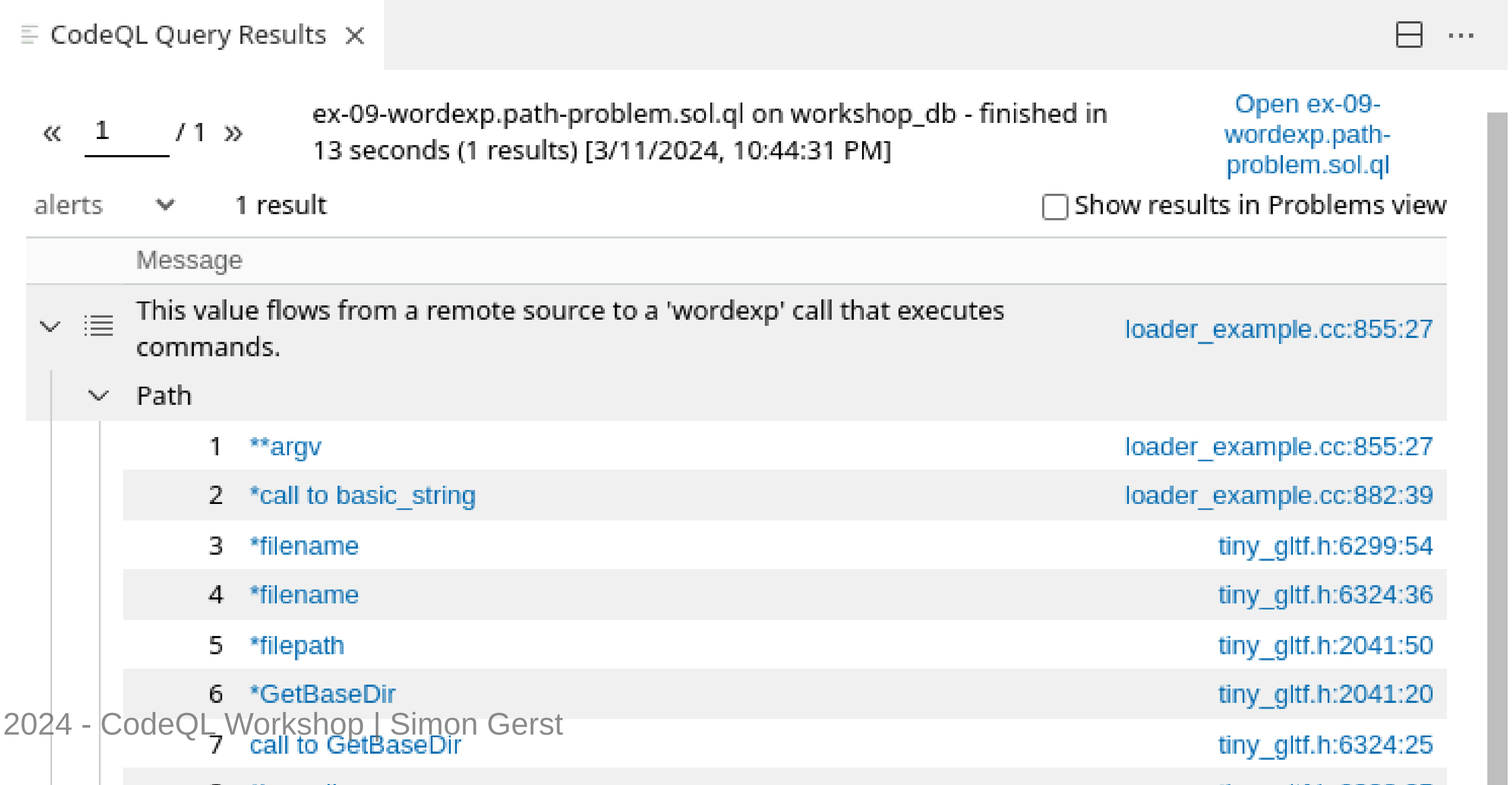
import cpp
import semmle.code.cpp.dataflow.new.TaintTracking
import semmle.code.cpp.security.FlowSources as FS
// CHANGED: Add this, so data flow queries "generate" results as a path.
import RemoteToWordexpFlow::PathGraph

module RemoteToWordexp implements DataFlow::ConfigSig {
  // unchanged [...]
}

// CHANGED: Instead of `DataFlow::Node`, we have to use `RemoteToWordexpFlow::PathNode`.
from RemoteToWordexpFlow::PathNode source, RemoteToWordexpFlow::PathNode sink
// CHANGED: Instead of `flow`, we have to use `flowPath`.
where RemoteToWordexpFlow::flowPath(source, sink)
select source, source, sink,
  "This value flows from a remote source to a 'wordexp' call that executes commands."
```

Putting Everything Together: Path-problem Solution

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



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.

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

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.

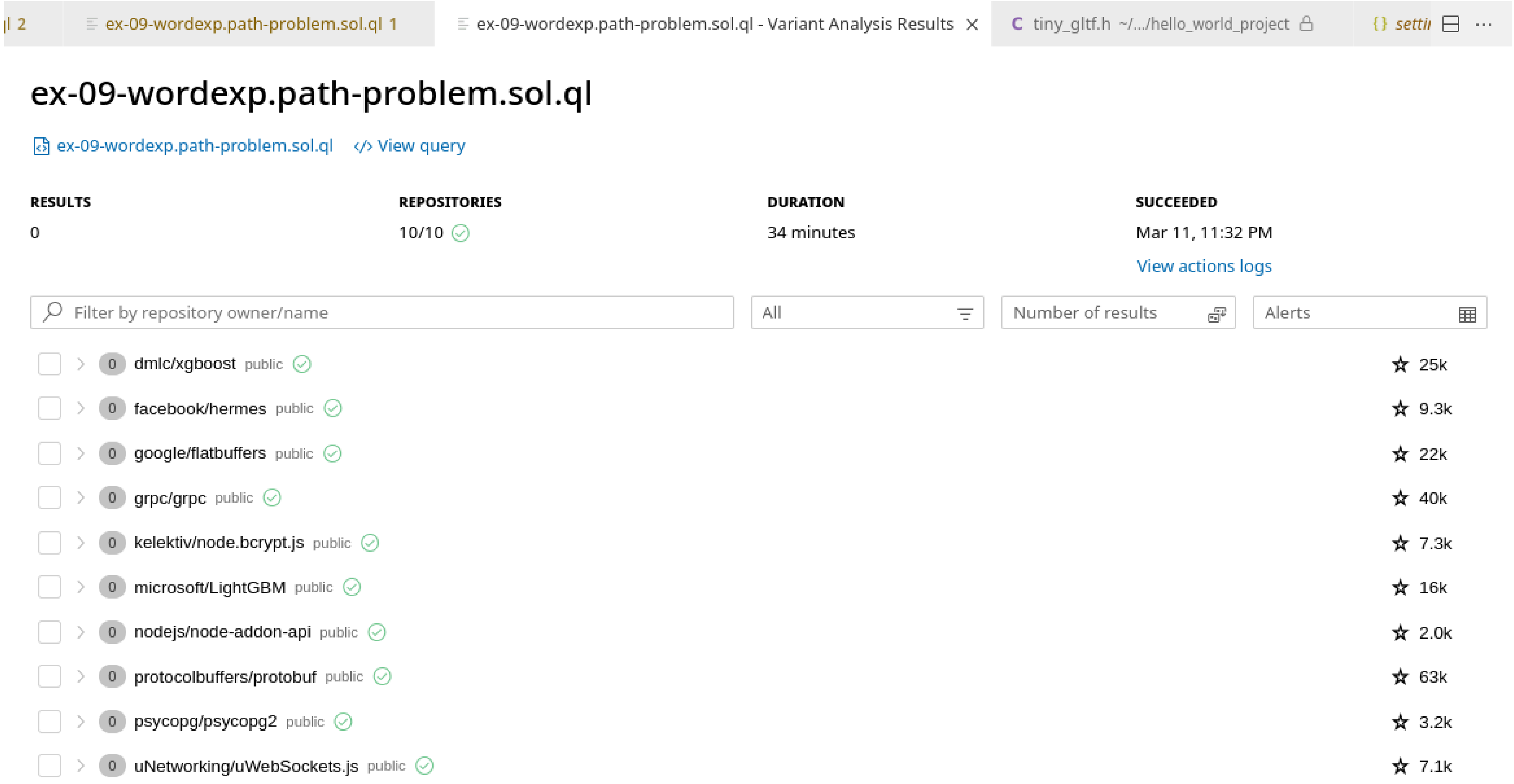
GitHub Codesearch

- Search for code across all of GitHub.
- Powerful regex queries.
- Can be used to find vulnerable code patterns:
    - Extract list of projects.
    - Run Multi-repository variant analysis (MRVA) on them.
    - Profit

Multi-repository variant analysis (MRVA)

- Run CodeQL queries across multiple (up to 1000) repositories.
- Security report found a vulnerable sink in your organization's code?
- Leverage MRVA to find all the places where remote data flows into these sinks.
- More information: Running CodeQL queries at scale with multi-repository variant analysis

MRVA results inside of VS Code



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 C/C++ 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.