
 **hohn** Full dataflow README, initial structure in using-db

c0e21f7 · 9 hours ago

 2

master	Go to file	t	Add file
exercises-tests	Full dataflow README, initial s...	9 hours ago	
exercises	Full dataflow README, initial s...	9 hours ago	
images	init	yesterday	
solutions-tests	Full dataflow README, initial s...	9 hours ago	
solutions	Full dataflow README, initial s...	9 hours ago	
using-db	Full dataflow README, initial s...	9 hours ago	
.gitignore	Full dataflow README, initial s...	9 hours ago	
LICENSE	init	yesterday	
README.html	init	yesterday	
README.md	Full dataflow README, initial s...	9 hours ago	
README.org	init	yesterday	
codeql-workspace.yml	Full dataflow README, initial s...	9 hours ago	
cpp-intro-i.code-works...	init	yesterday	

Readme Code of conduct MIT License Security policy

# CodeQL workshop for C/C++: dataflow I

In this workshop we will explore control flow, how it is represented by the standard library, and how you can use it to reason about reachability.

## Problem statement

In this workshop, we will use CodeQL to analyze the source code of [dotnet/coreclr](#), the runtime for .NET Core.

Formatting functions allow the programmer to construct a string output using a

```
ultimate goal:
select sink, source, sink, "This format string
may be derived from a $@", source, "user-
controlled value"
```

format string and an optional set of arguments. The format string is specified using a simple template language. The output string is constructed by processing the format string to find format specifiers, and inserting the values provided as arguments. For example, this `printf` statement: *simple query*

```
printf("Name: %s, Age: %d", "Freddie", 2);
```

would produce the output "Name: Freddie, Age: 2" . So far, so good. However, problems arise if there is a mismatch between the number of formatting specifiers, and the number of arguments. For example:

```
printf("Name: %s, Age: %d", "Freddie");
```

*mention*

In this case, we have one more format specifier than we have arguments. In a managed language such as Java or C#, this simply leads to a runtime exception. However, in C/C++, the formatting functions are typically implemented by reading values from the stack without any validation of the number of arguments. This means a mismatch in the number of format specifiers and the number of format arguments can lead to information disclosure (i.e. accessing unintended values from the top of the stack).

Of course, in practice this happens rarely with constant formatting strings. Instead, *it's most problematic when the formatting string can be specified by the user, allowing an attacker to provide a formatting string with the wrong number of format specifiers.* Furthermore, if an attacker can control the format string, they may be able to provide the `%n` format specifier, which causes `printf` to write the number characters in the generated output string to a specified memory location.

See [https://en.wikipedia.org/wiki/Uncontrolled\\_format\\_string](https://en.wikipedia.org/wiki/Uncontrolled_format_string) for more background.

This workshops will provide:

- Further experience writing real world queries
- Exploration of local data flow
- Exploration of local taint tracking
- Exploration of global data flow

## Contents

---

### Prerequisites and setup instructions

---

- Install [Visual Studio Code](#).
- Install the [CodeQL extension for Visual Studio Code](#).
- You do *not* need to install the CodeQL CLI: the extension will handle this for you.

- Clone this repository:

```
git clone https://github.com/hohn/codeql-dataflow-i-cpp
```

- Download the [dotnet/coreclr database](#).
- Import the unzipped database into Visual Studio Code:
  - Click the **CodeQL** icon in the left sidebar.
  - Place your mouse over **Databases**, and click the + sign that appears on the right.
  - Choose the downloaded database on your filesystem.
- Install the CodeQL pack dependencies using the command `CodeQL: Install Pack Dependencies` and select `exercises`, `exercises-tests`, `solutions`, and `solutions-tests`.

## Workshop

---

### Objectives

The workshop is split into multiple exercises introducing data flow. In these exercises you will learn:

- About data flow
- How data flow is represented in QL.
- About corner-cases when reasoning using control flow, how data flow provides higher level construct to answer reachability questions, but still requires control flow to excludes correct cases.

### Exercises

#### Exercise 1

Write a query that identifies `printf` (and similar) calls in the program. You can use the CodeQL standard library `semmlc.code.cpp.common.Printf` (by adding `import semmlc.code.cpp.common.Printf`).

Use the `FormattingFunctionCall` class, which represents many different formatting functions (`printf`, `snprintf` etc.).

```
from FormattingFunctionCall printfCall
select printfCall
```

#### Exercise 2

Update your query to report the "format string argument" to the `printf` -like call.

`printf` treats the first argument as the format string, however other formatting function calls (such as `snprintf`) pass the format string argument at a different position. For this reason, `FormattingFunctionCall` has a predicate named `getFormat()` which returns the format string argument, at whatever position it is.

```
from FormattingFunctionCall printfCall
select printfCall, printfCall.getFormat()
```

### Exercise 3

A simple way to find potentially risky `printf`-like calls is to report only the cases where the format argument is *not* a string literal. Refine the query to do this.

In the CodeQL standard library for C/C++ string constants are represented with the class `StringLiteral`. You can check whether a value is in a class using `instanceof`, for example `v instanceof StringLiteral`.

```
/**
 * @name Non-constant format string
 * @id cpp/non-constant-format-string
 * @kind problem
 */
import cpp
import semmle.code.cpp.common.Printf

from FormattingFunctionCall printfCall
where not printfCall.getFormat() instanceof StringLiteral
select printfCall.getFormat(), "Non-constant format argument"

// import cpp
// from FormattingFunctionCall
// printfCall
// where not printfCall.getFormat()
// instanceof StringLiteral
// select printfCall.getFormat(),
// "Non-constant format argument",
// printfCall.getLocation().getFile().getAbsolutePath()
```

### A Problem

Unfortunately, the results at this stage are unsatisfactory. For example, consider the results which appear in `/src/ToolBox/SOS/Strike/util.h`, between lines 965 and 970:

```
const char *format = align == AlignLeft ? "%-*.s" : "%*.s";

if (IsDMLEnabled())
    DMLOut(format, width, precision, mValue);
else
    ExtOut(format, width, precision, mValue);
```

Here, `DMLOut` and `ExtOut` are macros that expand to formatting calls. The format specifier is not constant, in the sense that the format argument is not a string literal. However, it is clearly only one of two possible constants, both with the same number of format specifiers.

What we need is a way to determine whether the format argument is ever set to something that is not constant.

## Data flow

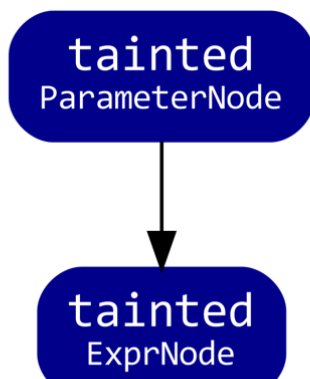
The solution here is to use data flow. Data flow is, as the name suggests, about tracking the flow of data through the program. It helps answers questions like: does this expression ever hold a value that originates from a particular other place in the program?

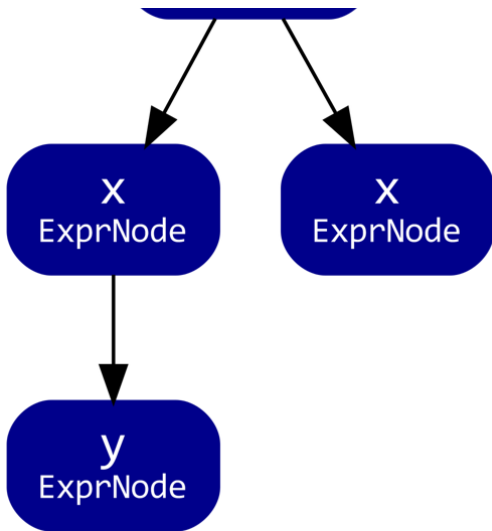
We can visualize the data flow problem as one of finding paths through a directed graph, where the nodes of the graph are elements in program, and the edges represent the flow of data between those elements. If a path exists, then the data flows between those two nodes.

Consider this example C function:

```
int func(int tainted) {  
    int x = tainted;  
    if (someCondition) {  
        int y = x;  
        callFoo(y);  
    } else {  
        return x;  
    }  
    return -1;  
}
```

The data flow graph for this function will look something like this:





This graph represents the flow of data from the tainted parameter. The nodes of graph represent program elements that have a value, such as function parameters and expressions. The edges of this graph represent flow through these nodes.

There are two variants of data flow available in CodeQL:

- Local ("intra-procedural") data flow models flow within one function; feasible to compute for all functions in a CodeQL database.
- Global ("inter-procedural") data flow models flow across function calls; not feasible to compute for all functions in a CodeQL database.

These are different APIs, so they will be discussed separately.

## Local dataflow

The data flow library can be imported from `semmle.code.cpp.dataflow.DataFlow`.

Note, this library contains an explicit "module" declaration:

```

import semmle.code.cpp.dataflow.DataFlow

module DataFlow {
  class Node extends ... { ... }
  predicate localFlow(Node source, Node sink) {
    localFlowStep*(source, sink)
  }
  ...
}
```

So all references will need to be qualified (that is, `DataFlow::Node`, rather than just `Node`).

The *nodes* are represented by the class `DataFlow::Node`. The data flow graph is separate from the AST (Abstract Syntax Tree, which represents the basic structure of the program), to allow for flexibility in how data flow is modeled. There are a small number of data flow node types – expression nodes, parameter nodes, uninitialized variable nodes, and definition by reference nodes, amongst others.

Each node provides mapping predicates to and from the relevant AST node (for example `Expr`, `Parameter` etc.) or symbol table (for example `Parameter`) classes. For example:

- `Node.asExpr()` asks for the AST `Expr` that is represented by this data flow node (if one exists)
- `Node.asParameter()` asks for the AST `Parameter` that is represented by this data flow node (if one exists)
- `DataFlow::exprNode(Expr e)` is a predicate that reports the `DataFlow::Node` for the given `Expr`.
- `DataFlow::parameterNode(Parameter p)` is a predicate that reports the `DataFlow::Node` for the given `Parameter`.

Similar helper predicates exist for the other types of node, although expressions and parameters are the most common.

The `DataFlow::Node` class is shared between both the local and global data flow graphs. The primary difference is the edges, which in the "global" case can link different functions.

The *edges* are represented by predicates. `DataFlow::localFlowStep` is the "single step" flow relation – that is, it describes single edges in the local data flow graph. `DataFlow::localFlow` represents the transitive closure of this relation – in other words, it contains every pair of nodes where the second node is reachable from the first in the data flow graph (i.e. a "reachability" table).

Consider again our C example, this time with the nodes numbered:

```
int func(int tainted) { // 1
    int x = tainted; // 2
    if (someCondition) {
        int y = x; // 3
        callFoo(y); // 4
    } else {
        return x; // 5
    }
    return -1;
}
```

Then the table of values computed for the `DataFlow::localFlowStep` predicate for this function will be:

nodeFrom	nodeTo
1	2
2	3

2	5
3	4

And the table of values computed for the `DataFlow::localFlow` predicate for this function will be:

source	sink
1	1
1	2
1	3
1	4
1	5
2	2
2	3
2	4
2	5
3	3
3	4
4	4
5	5

## Local data flow non-constant format string

One of the problems with our first query was that it did not exclude cases where the format string was constant but declared earlier in the function. We will now try to write a more refined version of our non-constant format string query by using local data flow to track back to the "source" of a format string within the function. We can then determine whether the "source" is a constant (i.e. string literal).

### Exercise 4

We will treat any data flow node without incoming local flow edges as a "source node" of interest. Define a predicate `isSourceNode` which identifies nodes without any incoming flow edges.

- Create a predicate called `isSourceNode` with a single argument called `sourceNode` of type `DataFlow::Node`.



- The edges in the data flow graph are represented by the predicate `DataFlow::localFlowStep`. You can think of this as providing a table of `nodeFrom, nodeTo` pairs for each edge in the graph. A node without any incoming flow edges will have no rows in the `DataFlow::localFlowStep` table where it is the second argument.
- You can use `not exists(..)` to state that there does not exist a `DataFlow::Node` that is the `nodeFrom` for the `sourceNode` parameter of your predicate.
- Alternatively, you can use `_` as a placeholder in any predicate call where you don't care about one of the values. This will allow you to avoid an explicit `exists`.

```
predicate isSourceNode(DataFlow::Node sourceNode) {
    // explicit version:
    not exists(DataFlow::Node nodeFrom |
        DataFlow::localFlowStep(nodeFrom, sourceNode)
    )
}
```

data may come  
from somewhere  
globally, but  
not locally

or more concisely

```
predicate isSourceNode(DataFlow::Node sourceNode) {
    not DataFlow::localFlowStep(_, sourceNode)
}
```

## Exercise 5

For this query it will be more convenient if we **define a CodeQL class for "source nodes"**. Define a new subclass of `DataFlow::Node` called `SourceNode`, and convert your predicate into the characteristic predicate.

- Create your own class `SourceNode` which extends `DataFlow::Node`.
- This will initially contain the set of all data flow nodes in the program. Add a characteristic predicate (`SourceNode() { ... }`) which restricts it to only the nodes without any incoming flow edges. Remember you can use the implicit variable `this`.

```
class SourceNode extends DataFlow::Node {
    SourceNode() {
        not DataFlow::localFlowStep(_, this)
    }
}
```

## Exercise 6

**Update the non-constant format string query to use local data flow and your new `SourceNode` class to filter out any format string calls where the source is a**

**SourceNode** class to filter out any format string calls where the source is a **StringLiteral**.

- Add a new QL variable called `src` with type `SourceNode`.
- Add a new QL variable called `formatStringArg` with type `DataFlow::Node`.
- Use `localFlow` to say that there is flow between the two.
- Use `asExpr()` to assert that the `getFormat()` expression of the `FormattingFunctionCall` is represented by the `formatStringArg` data flow node.
- Use `asExpr()` to assert that the *expression* represented by `src` is not a `StringLiteral`.

goal:

```
select formatStringArg, "Non-constant  
format string from $@.", src,  
src.toString()
```

```
/**  
 * @name Non-constant format string  
 * @id cpp/non-constant-format-string  
 * @kind problem  
 */  
import cpp  
import semmle.code.cpp.dataflow.DataFlow  
import semmle.code.cpp.common.Printf  
  
class SourceNode extends DataFlow::Node {  
  SourceNode() {  
    not DataFlow::localFlowStep(_, this)  
  }  
}
```

```
/**  
 * @name Non-constant format string  
 * @id cpp/non-constant-format-string  
 * @kind problem  
 */  
import cpp  
import semmle.code.cpp.dataflow.DataFlow  
import semmle.code.cpp.common.Printf  
  
class SourceNode extends DataFlow::Node {  
  SourceNode() {  
    not DataFlow::localFlowStep(_, this)  
  }  
}  
  
from FormattingFunctionCall printfCall,  
SourceNode src, DataFlow::Node formatStringArg  
where  
  formatStringArg.asExpr() =  
  printfCall.getFormat() and  
  DataFlow::localFlow(src, formatStringArg) and  
  not src.asExpr() instanceof StringLiteral  
select formatStringArg, "Non-constant format  
string from $@.", src, src.toString()
```

```
from FormattingFunctionCall printfCall, SourceNode src, DataFlow::Node  
where  
  formatStringArg.asExpr() = printfCall.getFormat() and  
  DataFlow::localFlow(src, formatStringArg) and  
  
  not src.asExpr() instanceof StringLiteral  
select formatStringArg, "Non-constant format string from $@.", src, src
```

## Local taint tracking

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. For example, `strcat` in the following

```
strcat(formatString, tainted); // source -> sink: taint flow step, not  
printf(formatString, ...)
```

would not be found by data flow alone because `strcat` 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

extends the standard data flow graph for a problem by adding edges between nodes where one node influences or taints another.

The taint-tracking API is almost identical to that of the local data flow. All we need to do to switch to taint tracking is `import semmle.code.cpp.dataflow.TaintTracking` instead of `semmle.code.cpp.dataflow.DataFlow`, and instead of using `DataFlow::localFlow`, we use `TaintTracking::localTaint`.

**Exercise 7** Update the non-constant format string query from the previous query to use local taint tracking instead of local data flow.

- Replace the data flow import with the taint tracking import
- Update the `SourceNode` class to refer to `TaintTracking::localTaintStep`.
- Update the query to refer to `TaintTracking::localTaint` instead of `DataFlow::localFlow`.

```
/**
 * @name Non-constant format string
 * @id cpp/non-constant-format-string
 *
 * @kind problem
 */
import cpp
import semmle.code.cpp.dataflow.TaintTracking
import semmle.code.cpp.common.Printf

class SourceNode extends DataFlow::Node {
  SourceNode() {
    not TaintTracking::localTaintStep(_, this)
  }
}

from FormattingFunctionCall printfCall, SourceNode src, DataFlow::Node
where
  formatStringArg.asExpr() = printfCall.getFormat() and
  TaintTracking::localTaint(src, formatStringArg) and
  not src.asExpr() instanceof StringLiteral
select formatStringArg, "Non-constant format string from $@.", src, sr
```

## Another Problem

Unfortunately, the results are still underwhelming. If we look through, we find that the main problem is that we don't know where the data originally comes from outside of the function, and there are an implausibly large number of results. One option would be to expand our query to track back through function calls, but handling parameters explicitly is cumbersome.

Instead, we could turn the problem around, and instead find user-controlled data that flows to a `printf` format argument, potentially through calls. This would be a global data flow problem, which we cover next.

# Introduction to global data flow

---

Many security problems can be phrased in terms of *information flow*:

*Given a (problem-specific) set of sources and sinks, is there a path in the data flow graph from some source to some sink?*

Some real world examples include:

- SQL injection: sources are user-input, sinks are SQL queries
- Reflected XSS: sources are HTTP requests, sinks are HTTP responses

We can solve such problems using the data flow and taint tracking libraries. Recall from the previous workshop that there are two variants of data flow available in CodeQL:

- Local ("intra-procedural") data flow models flow within one function
- Global ("inter-procedural") data flow models flow across function calls

While local data flow is feasible to compute for all functions in a CodeQL database, global data flow is not. This is because the number of paths becomes exponentially larger for global data flow.

The global data flow (and taint tracking) library avoids this problem by requiring that the query author specifies which *sources* and *sinks* are applicable. This allows the implementation to compute paths only between the restricted set of nodes, rather than for the full graph.

In this workshop we will try to write a global data flow query to solve the format string problem introduced in the [Local dataflow](#) section

The next sections provide:

- An exploration of global data flow and taint tracking
- Path queries
- Data flow sanitizers and barriers
- Data flow extra taint steps
- Data flow models

## Workshop

---

The workshop is split into several steps. You can write one query per step, or work with a single query that you refine at each step. Each step describes useful classes and predicates in the CodeQL standard libraries for C/C++. You can explore these in your IDE using the autocomplete suggestions (Ctrl + Space) and the jump-to-definition command (F12).

### Non-constant format strings

In the previous workshop we wrote a query to find non-constant format strings using local data flow:

```
/**
 * @name Non-constant format string
 * @id cpp/non-constant-format-string
 * @kind problem
 */
import cpp
import semmle.code.cpp.dataflow.TaintTracking
import semmle.code.cpp.common.Printf

class SourceNode extends DataFlow::Node {
  SourceNode() {
    not TaintTracking::localTaintStep(_, this)
  }
}

from FormattingFunctionCall printfCall, SourceNode src, DataFlow::Node
where
  formatStringArg.asExpr() = printfCall.getFormat() and
  TaintTracking::localTaint(src, formatStringArg) and
  not src.asExpr() instanceof StringLiteral
select formatStringArg, "Non-constant format string from $@.", src, sr
```

Unfortunately, the results are a little bit disappointing. For example, consider the result which appears in `/src/debug/createdump.cpp`, between lines 9 and 19:

```
//
// The common create dump code
//
bool
CreateDumpCommon(const char* dumpPathTemplate, MINIDUMP_TYPE minidumpT
{
  ReleaseHolder<DumpWriter> dumpWriter = new DumpWriter(*crashInfo);
  bool result = false;

  ArrayHolder<char> dumpPath = new char[PATH_MAX];
  snprintf(dumpPath, PATH_MAX, dumpPathTemplate, crashInfo->Pid());

  from Function f
  where
  f.hasName("CreateDumpCommon")
  select f
```

A parameter `dumpPathTemplate` is used as the format string in an `snprintf` call. However, we simply don't know whether the value of `dumpPathTemplate` can be set to something which is not constant, without looking at every caller of the method `CreateDumpCommon`.

## Untrusted data

If we consider what we mean by "non-constant", what we really care about is whether an attacker can provide this format string. As a first step, we will try to identify elements in the codebase which represent "sources" of untrusted user

goal 1: sources of untrusted input

goal 2: sinks: unchecked use of input

goal 3: connect them

input.

```
like
while (read(fd, &auxvEntry, sizeof(elf_aux_entry))
== sizeof(elf_aux_entry))
```

**Exercise 1:** Use the `semmle.code.cpp.security.Security` library to identify

`DataFlow::Node`s which are considered user input.

explore the class

- The `Security` library contains a `SecurityOptions` class. This class is a little different from the existing classes we have seen, because it is a *configuration* class. It does not represent any set of particular values in the database. Instead there is a single instance of this class, and it exists to provide predicates that are useful in security queries in a way that can be configured by the query writer.
- For our case it provides a predicate called `isUserInput`, which is a set of expressions in the program which are considered user inputs. You can use F12 to jump-to-definition to see what we consider to be a user input.
- `isUserInput` has two parameters, but we are not interested in the second parameter, so we can use `_` to ignore it.
- We want the `DataFlow::Node` for the user input, so remember to use `.asExpr()` to make that compatible with the `isUserInput` API.
- You can configure custom sources by creating a new subclass of `SecurityOptions`, and overriding one of the existing predicates. For today, the default definitions are sufficient.

```
import cpp
import semmle.code.cpp.dataflow.DataFlow
import semmle.code.cpp.security.Security

from SecurityOptions opts, DataFlow::Node source
where opts.isUserInput(source.asExpr(), _)
select source
```

## Format string injection

We now know where user input enters the program, and we know where format strings are used. We now need to combine these to find out whether data flows from one of these sources to one of these format strings. This is a global data flow problem.

As we mentioned in the introduction, the difference between local and global data flow is that we need to specify up front what sources and sinks we are interested in. This is because building a full table for a `globalTaint` predicate (i.e. the reachability table for the whole program), is just not feasible for any reasonable size of program, because it is bounded by  $O(n*n)$ , where  $n$  is the number of nodes.

The way we configure global data flow is by creating a custom extension of the `DataFlow::Configuration` class, and specifying `isSource` and `isSink` predicates:

```

class MyConfig extends DataFlow::Configuration {
  MyConfig() { this = "<some unique identifier>" }
  override predicate isSource(DataFlow::Node nd) { ... }
  override predicate isSink(DataFlow::Node nd) { ... }
}

```

Like the `SecurityOptions` class, `DataFlow::Configuration` is another *configuration* class. In this case, there will be a single instance of your class, identified by a unique string specified in the characteristic predicate. We then override the `isSource` predicates to represent the set of possible sources in the program, and `isSink` to represent the possible set of sinks in the program.

To use this new configuration class we can call the `hasFlow(source, sink)` predicate, which will compute a reachability table between the defined sources and sinks. Behind the scenes, you can think of this as performing a graph search algorithm from sources to sinks.

As with local data flow, we also have a *taint tracking* variant of global data flow. The API is almost identical, the main difference is that we extend

`TaintTracking::Configuration` instead. For the rest of this workshop we will use taint tracking, rather than data flow, as for this problem we really care about whether a user can influence a format string, not just whether they can specify the whole format string. This is because as long as an attacker can control part of the format string, they can usually place as many format specifiers as they like.

Here is an outline of the query we will write:

```

/**
 * @name Format string injection
 * @id cpp/format-string-injection
 * @kind problem
 */
import cpp
import semmle.code.cpp.dataflow.TaintTracking
import semmle.code.cpp.security.Security

class TaintedFormatConfig extends TaintTracking::Configuration {
  TaintedFormatConfig() { this = "TaintedFormatConfig" }
  override predicate isSource(DataFlow::Node source) {
    /* TBD */
  }
  override predicate isSink(DataFlow::Node sink) {
    /* TBD */
  }
}

from TaintedFormatConfig cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select sink, "This format string may be derived from a $@.", source, "

```

**Exercise 2: Implement the `isSource` predicate in this using the previous query for identifying user input `DataFlow::Node` s.**

- Use an `exists(..)` to introduce a new variable for `SecurityOptions` within the `isSource` predicate.

```

starting from
override predicate isSource(DataFlow::Node source) {
  exists (SecurityOptions opts |
    opts.isUserInput(source.asExpr(), _)
  )
}
override predicate isSource(DataFlow::Node source) {
  exists (SecurityOptions opts |
    opts.isUserInput(source.asExpr(), _)
  )
}

```

**Exercise 3: Implement the `isSink` predicate to identify format strings of format calls as sinks.**

- Use an `exists` to introduce a new variable of type `FormattingFunctionCall`, and use the predicate `getFormat()` to identify the format strings.
- Remember to use `DataFlow::Node.asExpr()` when comparing with the result of `getFormat()`.

Sink predicate:

```

from
import cpp

from FormattingFunctionCall printfCall
where not printfCall.getFormat()
instanceof StringLiteral
select printfCall.getFormat(), "Non-constant format argument"

override predicate isSink(DataFlow::Node sink) {
  exists (FormattingFunctionCall printfCall |
    sink.asExpr() = printfCall.getFormat()
  )
}

```

Full query:

```

import cpp
import semmlle.code.cpp.dataflow.TaintTracking
import semmlle.code.cpp.security.Security

class TaintedFormatConfig extends TaintTracking::Configuration {
  TaintedFormatConfig() { this = "TaintedFormatConfig" }
  override predicate isSource(DataFlow::Node source) {
    exists (SecurityOptions opts |
      opts.isUserInput(source.asExpr(), _)
    )
  }
  override predicate isSink(DataFlow::Node sink) {
    exists (FormattingFunctionCall printfCall |
      sink.asExpr() = printfCall.getFormat()
    )
  }
}

from TaintedFormatConfig cfg, DataFlow::Node
source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select sink, "This format string may be derived
from a $@.", source, "user-controlled value"

```



```

        sink.asExpr() = printfCall.getFormat()
    )
}
}

from TaintedFormatConfig cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select sink, "This format string may be derived from a $@.", source, "user-controlled value"

```

If we now run the query, we now only get 3 results, which is much more plausible! If we look at the results, however, it can be hard to verify whether the results are genuine or not, because we only see the source of the problem, and the sink, not the path in between.

## Path queries

The answer to this is to convert the query to a *path problem* query. There are five parts we will need to change:

- Convert the @kind from problem to path-problem. This tells the CodeQL toolchain to interpret the results of this query as path results.
- Add a new import DataFlow::PathGraph, which will report the path data alongside the query results.
- Change source and sink variables from DataFlow::Node to DataFlow::PathNode, to ensure that the nodes retain path information.
- Use hasFlowPath instead of hasFlow.
- Change the select to report the source and sink as the second and third columns. The toolchain combines this data with the path information from PathGraph to build the paths.

The format string injection as a path query:

```

/**
 * @name Format string injection
 * @id cpp/format-string-injection
 * @kind path-problem
 */

import cpp
import semmle.code.cpp.dataflow.TaintTracking
import semmle.code.cpp.security.Security
import DataFlow::PathGraph

class TaintedFormatConfig extends TaintTracking::Configuration {
    TaintedFormatConfig() { this = "TaintedFormatConfig" }
    override predicate isSource(DataFlow::Node source) {
        exists (SecurityOptions opts |
            opts.isUserInput(source.asExpr(), _)
        )
    }
    override predicate isSink(DataFlow::Node sink) {
        exists (FormattingFunctionCall printfCall |
            sink.asExpr() = printfCall.getFormat()
        )
    }
}

from TaintedFormatConfig cfg, DataFlow::PathNode source,
DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink, source, sink, "This format string may be
derived from a $@.", source, "user-controlled value"

```

```

}
override predicate isSink(DataFlow::Node sink) {
  exists (FormattingFunctionCall printfCall |
    sink.asExpr() = printfCall.getFormat()
  )
}
}
}

from TaintedFormatConfig cfg, DataFlow::PathNode source, DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink, source, sink, "This format string may be derived from a $@"

```

## Additional data flow features optional here

Data flow and taint tracking configuration classes support a number of additional features that help configure the process of building and exploring the data flow path.

One such feature is a *barrier or sanitizer node*. In some cases you will want to ignore paths that go through a certain set of data flow nodes, because those nodes either make the data safe, or validate that the data is safe. You can specify this by implementing another predicate in your configuration class, called `isSanitizer` (known as `isBarrier` for the `DataFlow::Configuration`).

For example, string format vulnerabilities generally can't be exploited if the user input is converted to an integer. We can model calls to the standard library `atoi` C function as a sanitizer like this:

```

override predicate isSanitizer(DataFlow::Node sanitizer) {
  exists (FunctionCall fc |
    fc.getTarget().getName() = "atoi" and
    sanitizer.asExpr() = fc
  )
}

```

A similar feature - `isSanitizerGuard` - allows you to specify a condition which is considered to be a "safety" check. If the check passes, the value should be considered safe.

Another feature is adding additional taint steps. This is useful if you use libraries which are not modelled by the default taint tracking. You can implement this by overriding `isAdditionalTaintStep` predicate. This has two parameters, the `from` and the `to` node, and essentially allows you to add extra edges into the taint tracking or data flow graph.

For example, you can model flow from C++ `string` `s` to C-style strings using the `.c_str()` method with the following additional taint step.

```

override
predicate isAdditionalTaintStep(DataFlow::Node node1, DataFlow::Node node2) {
  node1.asExpr().c_str() == node2.asExpr()
}

```

```
exists(FunctionCall fc |  
    fc.getTarget().getName() = "c_str" and  
    fc.getQualifier() = node1.asExpr() and  
    fc = node2.asExpr()  
)  
}
```

current Java example in  
<https://github.com/hohn/codeql-dataflow-ii-java/blob/master/SqlInjection.ql>

## About

similar example will be in <https://github.com/hohn/codeql-dataflow-II-cpp>



*No description, website, or topics provided.*

 Branches

 Tags

 0 stars

 1 watching

 0 forks

## Releases

No releases published

[Create a new release](#)

## Packages

No packages published

[Publish your first package](#)

## Languages

● CodeQL 72.5%   ● C++ 23.4%   ● C 4.1%