Java 数据流分析

您可以使用 CodeQL 来跟踪通过 Java 程序使用的数据流。

关于这篇文章

本文描述了如何在 CodeQL 库中实现数据流分析，并提供了一些示例来帮助您编写自己的数据流查询。以下部分描述如何使用库进行本地数据流、全局数据流和污点跟踪。

有关数据流建模的更一般性介绍，请参阅关于数据流分析。

本地数据流

本地数据流是单个方法或可调用的数据流。本地数据流通常比全局数据流更容易、更快、更精确，并且足以满足许多查询。

使用本地数据流

本地数据流库位于模块 DataFlow 中，它定义了表示数据可以通过的任何元素的类节点。节点分为表达式节点（ExprNode）和参数节点（ParameterNode）。可以使用成员谓词 asExpr 和 ASAMETER 在数据流节点和表达式/参数之间进行映射：

```
class Node {

  /** Gets the expression corresponding to this node, if any. */

  Expr asExpr() { ... }


  /** Gets the parameter corresponding to this node, if any. */

  Parameter asParameter() { ... }


  ...

}
```

或者使用谓词 exprNode 和 parameterNode：

```
/**

 * Gets the node corresponding to expression `e`.

 */

ExprNode exprNode(Expr e) { ... }
```

```
/**
 * Gets the node corresponding to the value of parameter `p` at
function entry.
 */
ParameterNode parameterNode(Parameter p) { ... }
```

谓词 localFlowStep（Node nodeFrom，Node nodeTo）保持从 nodeFrom 到 nodeTo 的即时数据流边缘。您可以通过使用+和*运算符递归地应用谓词，或者使用预定义的递归谓词 localFlow，它相当于 localFlowStep*。
例如，可以在零个或多个本地步骤中找到从参数源到表达式接收器的流：

```
DataFlow::localFlow(DataFlow::parameterNode(source),
DataFlow::exprNode(sink))
```

使用本地污点跟踪

局部污点跟踪通过包含非值保持流步骤来扩展本地数据流。例如：

```
String temp = x;

String y = temp + ", " + temp;
```

如果 x 是一个受污染的字符串，那么 y 也是受污染的。
本地污点跟踪库位于污点跟踪模块中。与本地数据流一样，谓词 localTaintStep（DataFlow:：Node nodeFrom，DataFlow:：nodeTo）保持从 nodeFrom 到 nodeTo 的即时污染传播边缘。可以通过使用+和*运算符递归地应用谓词，也可以使用预定义的递归谓词 localTaint，它相当于 localTaintStep*。
例如，可以在零个或多个本地步骤中找到从参数源到表达式接收器的污点传播：

```
TaintTracking::localTaint(DataFlow::parameterNode(source),
DataFlow::exprNode(sink))
```

示例

此查询查找传递给新 FileReader（..）的文件名。

```
import java
```

```
from Constructor fileReader, Call call
where
  fileReader.getDeclaringType().hasQualifiedName("java.io",
"FileReader") and
  call.getCallee() = fileReader
select call.getArgument(0)
```

不幸的是，这只给出参数中的表达式，而不是可以传递给它的值。因此，我们使用本地数据流来查找流入参数的所有表达式：

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

from Constructor fileReader, Call call, Expr src
where
  fileReader.getDeclaringType().hasQualifiedName("java.io",
"FileReader") and
  call.getCallee() = fileReader and
  DataFlow::localFlow(DataFlow::exprNode(src),
DataFlow::exprNode(call.getArgument(0)))
select src
```

然后我们可以使源更具体，例如访问公共参数。查找传递给新的文件的读取器（

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

from Constructor fileReader, Call call, Parameter p
where
```

```
  fileReader.getDeclaringType().hasQualifiedName("java.io",
"FileReader") and

  call.getCallee() = fileReader and

  DataFlow::localFlow(DataFlow::parameterNode(p),
DataFlow::exprNode(call.getArgument(0)))

select p
```

调用不包含此格式的字符串的函数。

```
import java

import semmle.code.java.dataflow.DataFlow

import semmle.code.java.StringFormat


from StringFormatMethod format, MethodAccess call, Expr formatString

where

  call.getMethod() = format and

  call.getArgument(format.getFormatStringIndex()) = formatString and

  not exists(DataFlow::Node source, DataFlow::Node sink |

    DataFlow::localFlow(source, sink) and

    source.asExpr() instanceof StringLiteral and

    sink.asExpr() = formatString

  )

select call, "Argument to String format method isn't hard-coded."
```

练习

练习 1：编写一个查询，查找用于创建 java.net.URL，使用本地数据流。（回答）

全局数据流

全局数据流跟踪整个程序中的数据流，因此比本地数据流更强大。然而，全局数据流不如本地数据流精确，分析通常需要更多的时间和内存来执行。

注意
可以通过创建路径查询在 CodeQL 中对数据流路径进行建模。要查看由 codeqlforvs 代码中的路径查询生成的数据流路径，需要确保它具有正确的元数据和 select 子句。有关详细信息，请参见创建路径查询。

使用全局数据流

您可以通过扩展 DataFlow::Configuration 类来使用全局数据流库：

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


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


  override predicate isSource(DataFlow::Node source) {
    ...
  }


  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

这些谓词在配置中定义：

- isSource 定义数据的来源
- ISink 定义数据可能流向的位置
- isBarrier 可选，限制数据流
- isAdditionalFlowStep 可选，添加其他流步骤

特征谓词 MyDataFlowConfiguration（）定义配置的名称，因此“MyDataFlowConfiguration”应该是唯一的名称，例如，类的名称。
使用谓词 hasFlow（DataFlow:: Node source，DataFlow:: Node sink）执行数据流分析：

```
from MyDataFlowConfiguration dataflow, DataFlow::Node source,
DataFlow::Node sink

where dataflow.hasFlow(source, sink)
```

```
select source, "Data flow to $@.", sink, sink.toString()
```

使用全局污染跟踪

全局污点跟踪针对全局数据流，正如局部污点跟踪针对本地数据流一样。也就是说，全局污点跟踪通过附加的非保值步骤扩展了全局数据流。您可以通过扩展 TaintTracking:: Configuration 类来使用全局污染跟踪库：

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


class MyTaintTrackingConfiguration extends
TaintTracking::Configuration {

  MyTaintTrackingConfiguration() { this =
"MyTaintTrackingConfiguration" }


  override predicate isSource(DataFlow::Node source) {

    ...

  }


  override predicate isSink(DataFlow::Node sink) {

    ...

  }

}
```

这些谓词在配置中定义：

- isSource 定义了污点的来源
- isSink 定义了污点可能流向何处
- isSanitizer 可选，限制污染流
- isAdditionalTaintStep 可选，添加其他污染步骤

与全局数据流类似，特征谓词 MyTaintTrackingConfiguration（）定义配置的唯一名称。

污点跟踪分析使用谓词 hasFlow（DataFlow:: Node source，DataFlow:: Node sink）执行。

流动源

数据流库包含一些预定义的流源。类 RemoteFlowSource（在中定义 semmle.code.java.数据流.FlowSources)表示可能由远程用户控制的数据流源，这对于查找安全问题很有用。

示例

此查询显示使用远程用户输入作为数据源的污点跟踪配置。

```
import java

import semmle.code.java.dataflow.FlowSources


class MyTaintTrackingConfiguration extends
TaintTracking::Configuration {

  MyTaintTrackingConfiguration() {

    this = "..."

  }


  override predicate isSource(DataFlow::Node source) {

    source instanceof RemoteFlowSource

  }


  ...

}
```

练习

练习 2：编写一个查询，查找用于创建 java.net.URL，使用全局数据流。（回答）
练习 3：编写一个表示流源的类 java.lang.System.getenv（…）。（回答）
练习 4：使用 2 和 3 中的答案，编写一个查询，查找从 getenv 到 java.net.URL. （回答）

答案

练习 1

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


from Constructor url, Call call, StringLiteral src
where
  url.getDeclaringType().hasQualifiedName("java.net", "URL") and
  call.getCallee() = url and
  DataFlow::localFlow(DataFlow::exprNode(src),
DataFlow::exprNode(call.getArgument(0)))
select src
```

练习 2

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


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

  override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof StringLiteral
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      sink.asExpr() = call.getArgument(0) and
```

```
call.getCallee().(Constructor).getDeclaringType().hasQualifiedName("
java.net", "URL")

    )

  }

}


from DataFlow::Node src, DataFlow::Node sink, Configuration config

where config.hasFlow(src, sink)

select src, "This string constructs a URL $@.", sink, "here"
```

练习 3

```
import java


class GetenvSource extends MethodAccess {

  GetenvSource() {

    exists(Method m | m = this.getMethod() |

      m.hasName("getenv") and

      m.getDeclaringType() instanceof TypeSystem

    )

  }

}
```

练习 4

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


class GetenvSource extends DataFlow::ExprNode {
```

```
  GetenvSource() {
    exists(Method m | m = this.asExpr().(MethodAccess).getMethod() |
      m.hasName("getenv") and
      m.getDeclaringType() instanceof TypeSystem
    )
  }
}

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

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

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      sink.asExpr() = call.getArgument(0) and

call.getCallee().(Constructor).getDeclaringType().hasQualifiedName("
java.net", "URL")
    )
  }
}
```

```
from DataFlow::Node src, DataFlow::Node sink,
GetenvToURLConfiguration config

where config.hasFlow(src, sink)

select src, "This environment variable constructs a URL $@.", sink,
"here"
```

进一步阅读

- 用路径查询探索数据流
- Java 的 CodeQL 查询
- Java 查询示例
- Java 代码库参考
- QL 语言参考