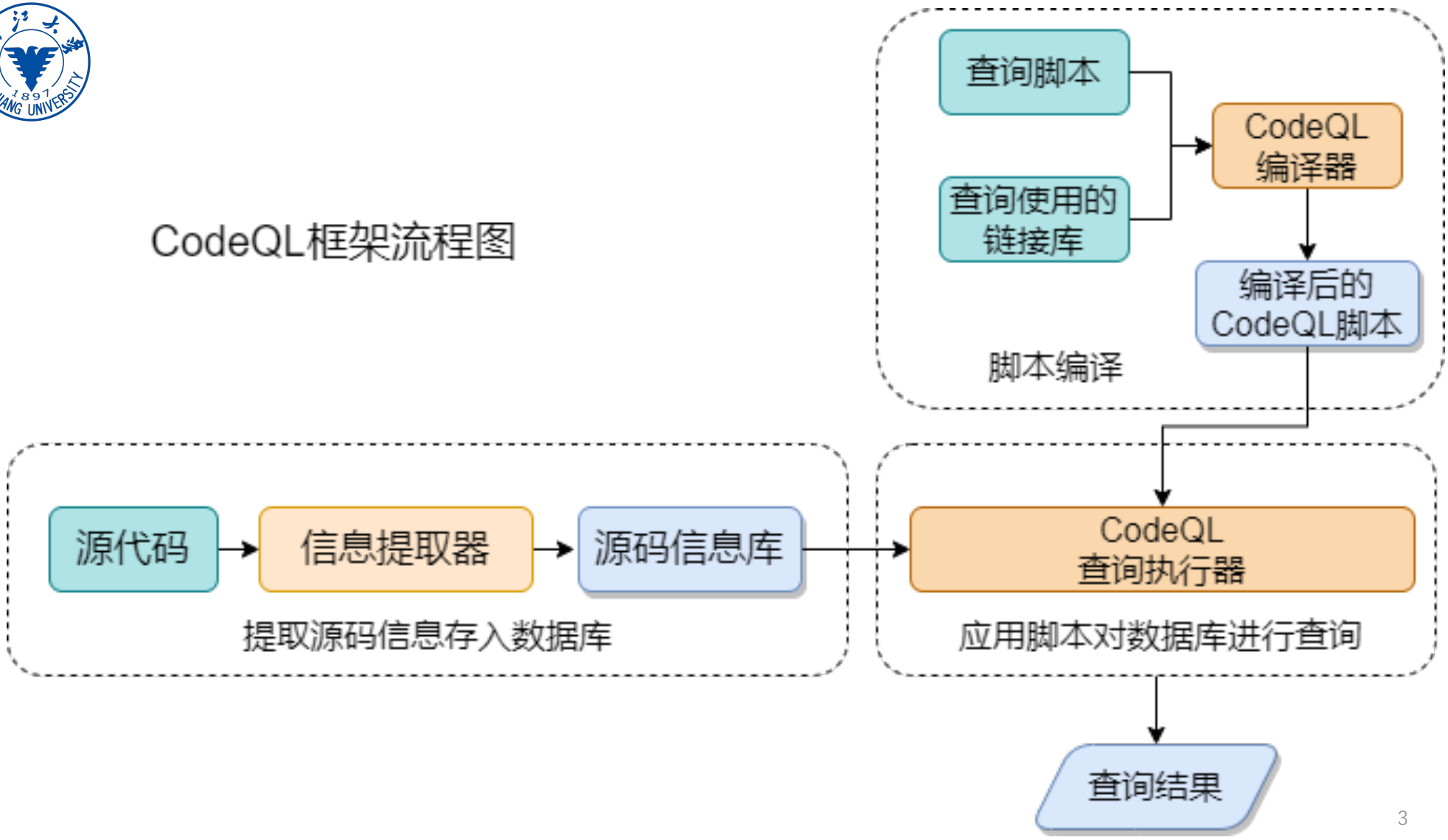
浙江大學
ZHEJIANG UNIVERSITY

静态分析工具-- CodeQL

CodeQL简介

- github安全团队开发的静态分析引擎
- 应用范围：变种分析(variant analysis)
- 原理：源代码的语法、语义、数据类型、数据流图、控制流图等相关信息被提取到数据库中，使代码可以被当作数据一样进行查询

CodeQL框架流程图



CodeQL常用功能模块

1. CodeQL数据流分析

➢ 对程序运行时的数据流向进行了建模
- 局部数据流分析(同一个函数内部的数据流向)
- 全局数据流分析(同一个函数内部、函数与函数之间的数据流向)

✓ **DataFlow模块**

2. CodeQL污点分析

➢ 三元组**<sources,sinks,sanitizers>**
- 污点源→污点汇聚点

✓ **TaintTracking模块**

CodeQL常用功能模块

3. CodeQL指向分析

➢ 在编译期间建立指针变量和在运行时它所指向的地址之间的关系
- 基于**Steensgaard**算法(过程间的流不敏感指向分析)

✓ **PointsTo**模块

4. CodeQL控制流分析

➢ 控制流图是对程序执行过程中控制流的静态过近似图形表示
- 基于抽象语法树实现

✓ **ControlFlow**模块(过程内的)

CodeQL语法

select...from...where...

- 面向对象(仅代表源代码在逻辑上的关系)

查询结构

```
/**
* Query metadata
*/

import /* ... CodeQL libraries or modules ... */

/* ... Optional, define CodeQL classes and predicates ... */

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

CodeQL语法

1. QL文件的元数据(Query metadata)

```
/**
 * @name Find bugs in source code
 * @description Find bugs about buffer overflow,format string and use after free.
 * @id cpp/example
 * @kind path-problem
 * @precision very-high
 * @tags security
 * @security-severity 6.0
 */
```

CodeQL语法

2. Import 声明

- 用于将**CodeQL**库或模块导入到查询中

可以使用官方提供的库或模块，也可以自定义查询库

3. From 子句

- 用于声明在查询中使用的变量

使用以下形式进行定义：**<type> <variable name>**

CodeQL中主要的变量类型：**boolean**、**float**、**int**、**string**、**date**

支持不同类型间的转化，也可以使用**class**定义自己的类型

CodeQL语法

4. Where子句

- 定义了应用于变量的逻辑条件，可以缩小查询出的结果范围。
- 逻辑条件可以用集合体**(aggregation)**、谓词**(predicate)**、逻辑公式**(logical formulas)**三种类型来定义。

- 集合体：**count**、**min**、**max**、**avg**、**sum**、**concat**、**rank**、**unique**等。

- 谓 词：用**predicate**关键词定义，用于描述程序各部分间的逻辑关系(相当于高级编程语言中的函数)，可以递归使用。

- 逻辑公式：**>**、**<**、**>=**、**<=**、 **=**、 **!=**、**instanceof**、 **in**、 **exists**、**not**、**if..then..else**、**and**、**or**、**implies**。

CodeQL语法

5. Select子句

- Select子句用于显示所有满足where子句所规定的条件的结果。结果的格式和元数据中@kind属性有关。

如果@kind为problem
遵循以下格式：select element, string

如果@kind为path-problem
遵循以下格式：select element, source, sink, string

几种常用查询对象类型

Function 指代程序中函数定义、函数声明
FunctionCall 指代函数调用

Macro 指代宏定义
MacroInvocation 指代宏调用

Expr 指代表达式
AssignExpr 指代赋值表达式（是Expr的子集）
ConditionalStmt 指代条件表达式

Field 指代结构体中的成员

CodeQL 示例

检测源码中是否存在system函数

源程序：

```
void process0(){
    printf("hello!\n");
    system("/bin/sh");
}
```

检测程序：

```
class System extends FunctionCall{
    System(){
        exists( FunctionCall fc |
            fc.getTarget().getName() = "system"|
            this = fc
        )
    //以上语句的含义为找到system()函数并存放this变量(代表当前System类)
    }
}

from System sys
select sys.getEnclosingFunction() as
//以上语句的含义为找到调用system()的函数的名称
function_name, sys.getLocation() as location
//以上语句的含义为找到system()函数的位置
```

检测缓冲区溢出漏洞

源程序：

```
#define LENGTH 10
char buffer[LENGTH]={'0'};

//buffer overflow
void process1() {
    char buf[LENGTH]={'0'};
    int length;
    scanf("input length: %d",&length);
    read(0,buf,length);
}
//correct version
void process2() {
    char buf[LENGTH]={'0'};
    unsigned int length;
    scanf("input length: %d",&length);
    if(length<LENGTH)
        read(0,buf,length);
}
```

process1()为具有缓冲区溢出漏洞的函数，没有对输入数据的长度进行检查。

process2()为正确的函数，对长度进行了条件检查。

检测缓冲区溢出漏洞

检测思路：

1.找到read()函数；

2.获取read()函数第3个参数(可输入数据的长度)；

3.判断输入数据的长度是否可能超过数组的最大上限。

检测缓冲区溢出漏洞

检测程序：

```
class Bof extends FunctionCall {
    Bof(){
        exists( FunctionCall fc, Expr length |
            fc.getTarget().getName() ="read" and
            length = fc.getArgument(2) and
            //以上语句的含义为找到read()第三个参数，即用户可输入的字符串长度。
            not exists( Macro v, LocalVariable var, int uppersize |
                uppersize = v.getBody().toInt() and v.getHead() = "LENGTH" |
                var.getName().matches("length") and upperBound(length) <= uppersize
            ) |
            //以上语句的含义为判断是否存在用户可输入的长度大于缓冲区的大小(LENGTH)
            this = fc //将查询到的函数存入this变量(代表当前bof类)
        )
    }
}
```

检测缓冲区溢出漏洞

检测程序：

```
from Bof bof
select bof.getEnclosingFunction() as function_name,
//以上语句含义为获取查询出的函数的调用者

bof.getLocation().toString() as location
//以上语句含义为获取查询出的函数的位置
```

检测缓冲区溢出漏洞演示

检测格式化字符串漏洞

源程序：

```
char buffer[LENGTH]={'0'};


//format string
void process3() {
    read(STDIN_FILENO, buffer, LENGTH);
    printf(buffer);
}
//correct version
void process4() {
    read(STDIN_FILENO, buffer, LENGTH);
    printf("buffer:%s\n", buffer);
}
```

process3()为具有格式化字符串漏洞的函数，没有对输入数据的长度进行检查。

process4()为正确的函数。

检测格式化字符串漏洞

检测思路：

1.找到printf()函数；

2.获取printf()函数格式化字符参数；

3.应用局部数据流分析，检查格式化字符参数是否为硬编码。

检测格式化字符串漏洞

检测程序：

```
class Format extends FunctionCall{
    Format(){
        exists(
            FormattingFunction format,FunctionCall call, Expr formatString |
            call.getTarget() = format
            and call.getArgument(format.getFormatParameterIndex()) = formatString
            //以上语句含义为查找调用格式化字符串的函数并且格式化字符串为函数的第一个参数
            and not exists( DataFlow::Node source, DataFlow::Node sink |
                DataFlow::localFlow(source, sink) and
                source.asExpr() instanceof StringLiteral and
                sink.asExpr() = formatString
                ) |
            //以上语句应用局部数据流分析查找是否存在格式化字符串不是硬编码的函数
            this = call
        )
    }
}
```

检测格式化字符串漏洞

检测程序：

```
from Format form
select form.getEnclosingFunction() as function_name,
//以上语句含义为获取查询出的函数的调用者

form.getLocation() as Location
//以上语句含义为获取查询出的函数的位置
```

检测格式化字符串漏洞演示

检测UAF漏洞

源程序：

```
struct MyStruct {
    char* buf;
};

static void writebuf(struct MyStruct *s, int id) {
  sprintf(s->buf, "buffer: %d\n", id);
}

//use after free
void process5() {
    struct MyStruct* s = (struct MyStruct*)malloc(sizeof(struct MyStruct));
    s->buf = malloc(LENGTH);
    read(STDIN_FILENO, s->buf, LENGTH);
    free(s->buf);
    writebuf(s,1);
}
```

释放s->buf后又通过writebuf进行了使用

检测UAF漏洞

检测思路：

1.找到free()函数的参数；

2.应用全局数据流分析，检查free()的参数是否会在其它地方被解引用。

检测UAF漏洞

检测程序：

```
//global data flow
class Analysis extends DataFlow::Configuration{
    Analysis() { this = "use-after-free" }
    override predicate isSource(DataFlow::Node arg) {
        exists( FunctionCall call |
            call.getArgument(0) = arg.asPartialDefinition() and
            call.getTarget().hasGlobalOrStdName("free")
        )
    }
    //以上语句的含义为找到free函数的第一个参数，并设置该参数为source
    override predicate isSink(DataFlow::Node sink) {
        dereferenced(sink.asExpr())
    }
    //以上语句的含义为找到解引用free函数参数的地方，设置为sink
}
```

检测UAF漏洞

检测程序：

```
from DataFlow::PathNode source,DataFlow::PathNode sink,Analysis uaf
where uaf.hasFlowPath(source, sink)
//以上语句的含义为找到从source到sink存在路径的对象

select source.getNode().getEnclosingCallable() as source_func,
//以上语句的含义为找到调用source点的函数

source.getNode().getLocation() as source_location,
//以上语句的含义为找到source点的位置

sink.getNode().getEnclosingCallable() as
//以上语句的含义为找到调用sink点的函数

sink_func,sink.getNode().getLocation() as sink_location
//以上语句的含义为找到sink点的位置
```