codeql文档

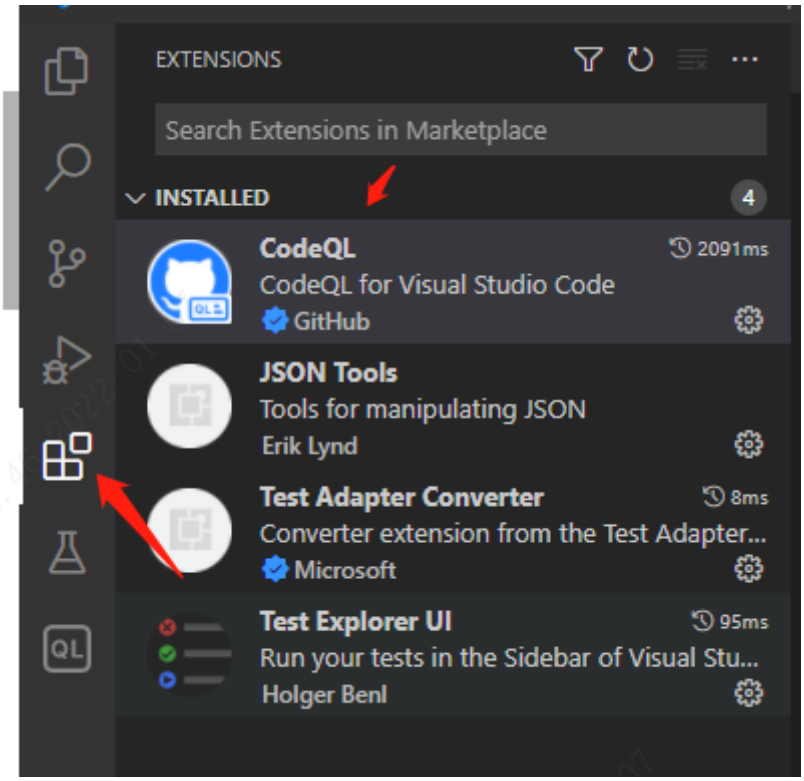
codeql配置

codeql分为引擎+SDK两部分,SDK内含有QL的语法

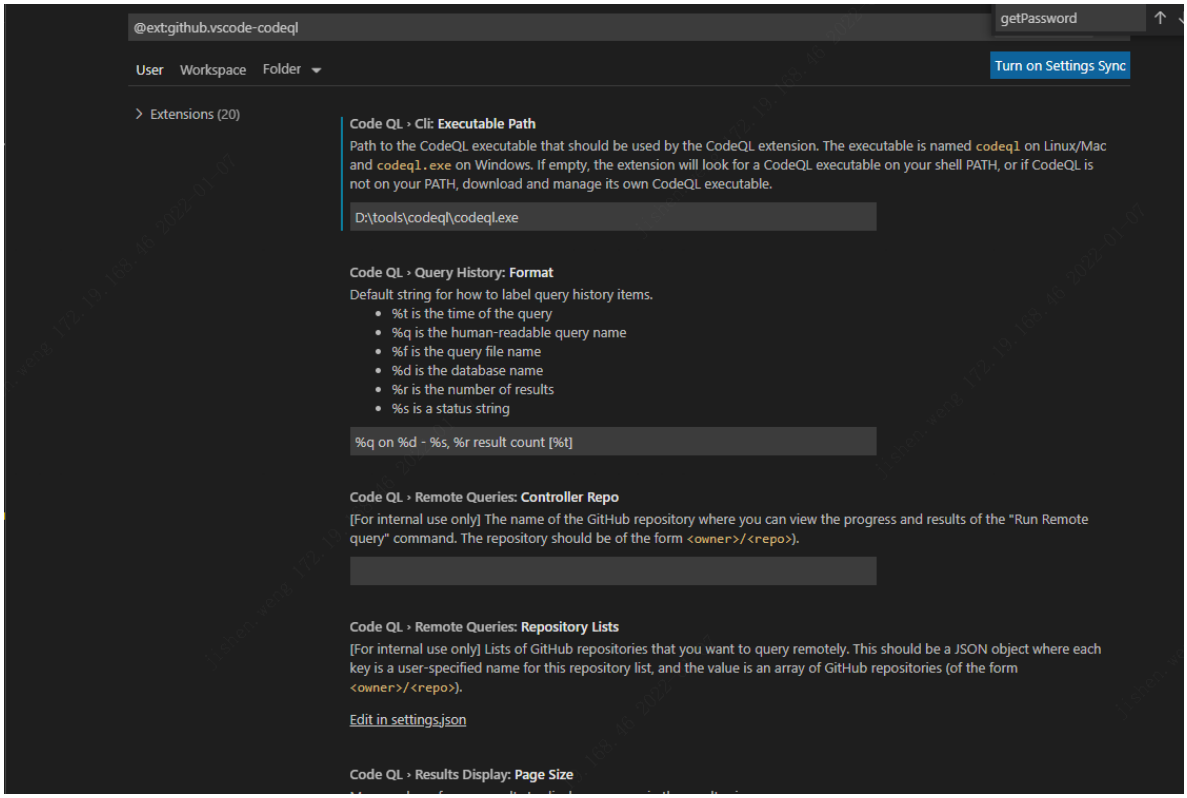
引擎下载地址: https://github.com/github/codeql-cli-binaries/releases

SDK下载地址: https://github.com/Semmle/ql

vscodeQL插件配置: 通过extensions安装QL插件



配置codeql.exe可执行目录



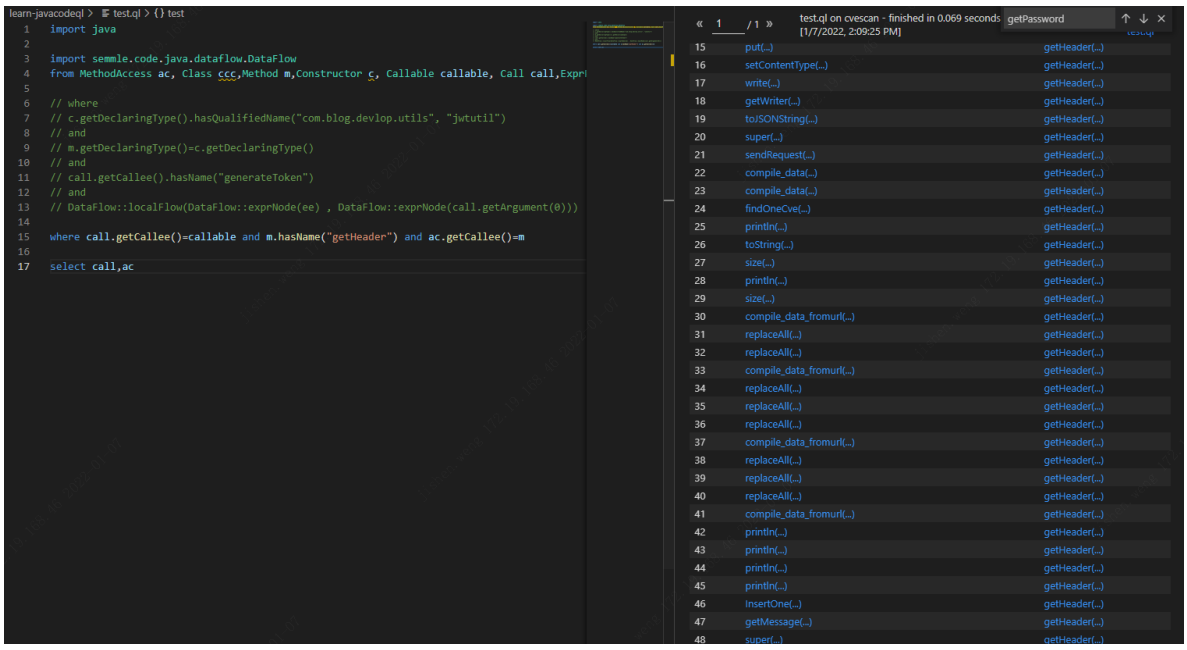
创建一个新文件夹,把sdk放入文件夹根目录中

codeql创建数据库

codeql根据用户提供不同语言的代码,利用codeql引擎分析代码之间的关系后生成相数据库,生成数据库后,导入数据库即可编辑自定义查询语言对数据库进行查询

```
codeql database create 数据库存储地址 --language="java"  --command="mvn clean install --file pom.xml" --source-root=项目地址
```

查询



codeql语法

基本语法

模块

对不同的语言查询,要导入相对应的Codeql库,codeql支持语言有 `c c++ c# Go Java Javascript python ruby`

该库中的类以面向对象的形式呈现数据库中的数据,并提供抽象和谓词帮助用户完成常见的分析任务

标准库可分为五个主要类别:

- 用于表示程序元素的类(类和方法)
- 表示AST节点的类(语句和表达式)
- 表示元数据的类(注释)
- 用于计算度量的类
- 用于导航程序调用图的类

导入例子

```
import java
```

程序元素

包含`类型 Type 方法 Method 构造函数 constructor 变量 variable 包 package`

类型 Type:

类型中包含许多子类,如PrimitiveType,Reftype

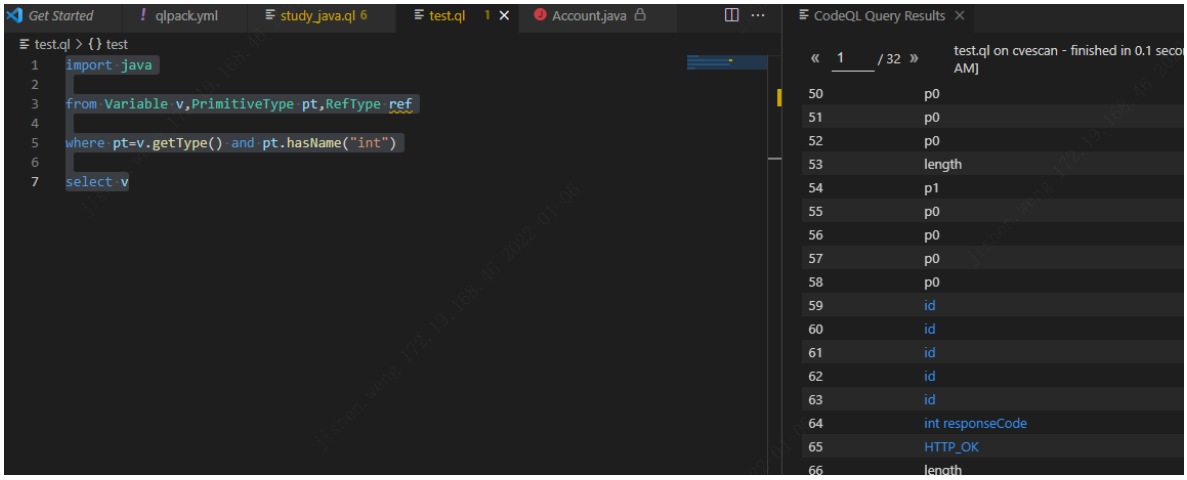
- PrimitiveType表示原始类型 包含`boolean byte char double float int long void null`
- Reftype表示引用类型:`Class interface enumtype array`

```
import java

from Variable v,PrimitiveType pt,RefType ref

where pt=v.getType() and pt.hasName("int")

select v
```

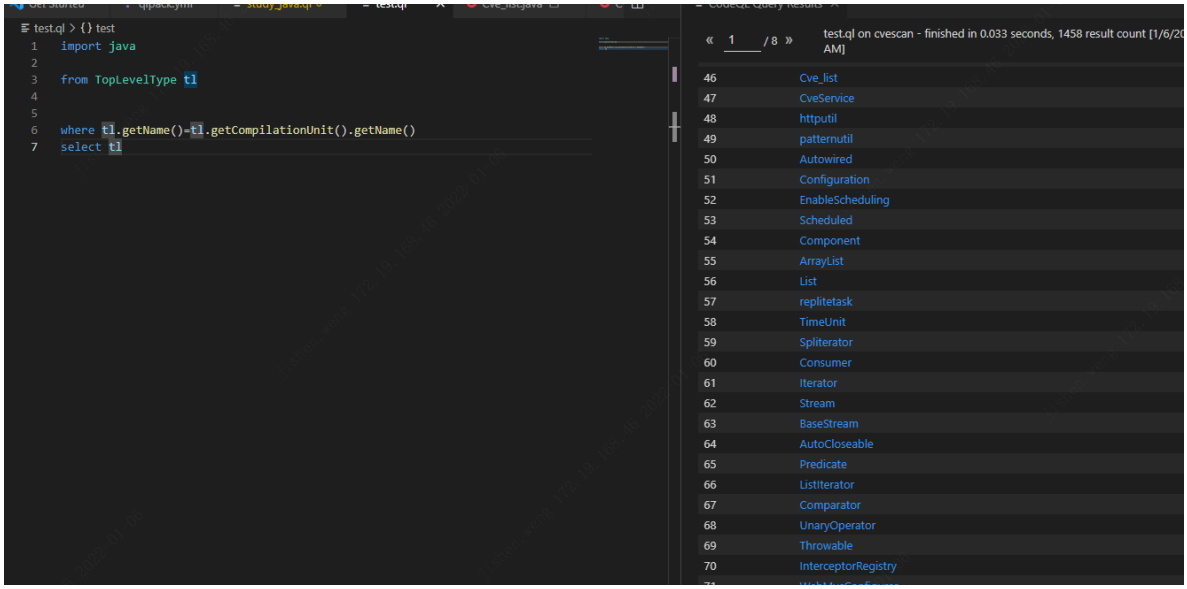


- TopLevelClass 表示顶层声明的类(含接口)

```
import java

from TopLevelType tl


where tl.getName()=tl.getCompilationUnit().getName()
select tl
```



NestedClass表示在另一个类中声明的类(内部类 匿名类)

变量:

- QL用以下模块来查询变量

```
Variable 常规变量
Field 类中定义的成员
LocalVariableDecl 表示局部变量
Paramter 表示方法或构造函数的参数
```

抽象语法树:

抽象语法树使用`Stmt`和表达式`expr`模块

- `Expr.getAChildExpr` 返回给定表达式的子表达式。
- `Stmt.getAChild` 返回直接嵌套在给定语句中的语句或表达式。
- `Expr.getParent`并`Stmt.getParent`返回 AST 节点的父节点。

- Stmt类型

Statement syntax	CodeQL class	Superclasses	Remarks
`;`	EmptyStmt		
`Expr ;`	ExprStmt		
`{ Stmt ... }`	BlockStmt		
`if ( Expr ) Stmt else Stmt`	IfStmt	ConditionalStmt	
`if ( Expr ) Stmt`			
`while ( Expr ) Stmt`	WhileStmt	ConditionalStmt, LoopStmt	
`do Stmt while ( Expr )`	DoStmt	ConditionalStmt, LoopStmt	
`for ( Expr ; Expr ; Expr ) Stmt`	ForStmt	ConditionalStmt, LoopStmt	
`for ( VarAccess : Expr ) Stmt`	EnhancedForStmt	LoopStmt	
`switch ( Expr ) { SwitchCase ... }`	SwitchStmt		
`try { Stmt ... } finally { Stmt ... }`	TryStmt		
`return Expr ;`	ReturnStmt		
`return ;`			
`throw Expr ;`	ThrowStmt		
`break ;`	BreakStmt	JumpStmt	
`break label ;`			
`continue ;`	ContinueStmt	JumpStmt	
`continue label ;`			
`label : Stmt`	LabeledStmt		
`synchronized ( Expr ) Stmt`	SynchronizedStmt		
`assert Expr : Expr ;`	AssertStmt		
`assert Expr ;`			
`TypeAccess name ;`	LocalVariableDeclStmt		
`class name { Member ... } ;`	LocalClassDeclStmt		
`this ( Expr , ... ) ;`	ThisConstructorInvocationStmt		
`super ( Expr , ... ) ;`	SuperConstructorInvocationStmt		
`catch ( TypeAccess name ) { Stmt ... }`	CatchClause		can only occur as child of a TryStmt
`case Literal : Stmt ...`	ConstCase		can only occur as child of a SwitchStmt
`default : Stmt ...`	DefaultCase		can only occur as child of a SwitchStmt

例子:

```
import java

from Expr e,Stmt st,TopLevelType tl,NestedClass nc,TypeVariable
type,GenericInterface map,ParameterizedType pt,TypeBound tb,Variable v,Field
f,LocalVariableDecl vv



where e.getParent() instanceof IfStmt //查询子表示为if语句类型


select e.getAChildExpr(),e.getParent()
```

数据流

数据流在分析中占到重中之重的地位,数据流分析可以查询 是否将危险参数传递给函数 或敏感数据是是否会泄露,除了突出潜在的安全问题外,还可以使用数据流来分析程序行为

codeql数据流分为三种:局部数据流 全局数据流 全局污点跟踪

数据流和污点跟踪区别:

普通数据流用于分析信息流,其中每个步骤都保留数据值

污点跟踪扩展了数据流分析, 包括不一定保留数据值但仍传播潜在不安全对象的步骤。这些流程步骤在污点跟踪库中使用谓词建模，如果污点在节点之间传播，则该谓词保持不变。

source/sink/sanitizer :

什么是source和sink

在自动化代码审计理论中,有一个核心的三个概念(source sink sanitizer)

source指漏洞污染的入口点,比如http请求参数,参数就是一个source

sink指的是漏洞出发点,比如sql注入,最终拼接sql语句并执行的地方就是一个sink

sanitizer 净化函数,在漏洞链条中,如果存在一个方法阻断了调用链,那么这个方法就叫 sanitizer

局部数据流

局部数据流用于查找单个方法内数据的流向,比全局数据流更快 更精准

局部数据流位于模块`DataFlow`,并定义了`Node`用于表示数据流经的类.Node分为表达式节点(`exprNode`)和参数节点(`ParamterNode`) 可以使用`asExpr`在数据流节点和表达式参数之间映射`asParameter`

局部数据流是通过使用`DataFlow::localFlow`谓词

局部数据流例子:

```
import java

import semmle.code.java.dataflow.DataFlow
from Class ccc,Method m,Constructor c, Call call,ExprParent ee, Expr e,Stmt
st,TopLevelType tl,NestedClass nc,TypeVariable type,GenericInterface
map,ParameterizedType pt,TypeBound tb,Variable v,Field f,LocalVariableDecl vv

where
c.getDeclaringType().hasQualifiedName("com.blog.devlop.utils", "jwtutil")  //限定
类名
and
m.getDeclaringType()=c.getDeclaringType() //方法的类名要等同于限定的类名
and
call.getCallee().hasName("generateToken") //限定方法名
and DataFlow::localFlow(DataFlow::exprNode(e) ,
DataFlow::exprNode(call.getArgument(0))) //查看是否能流通

select call
```

全局数据流

全局数据流在整个程序中跟踪数据流,因此比局部数据流强大,然而全局数据流不如局部数据流精准

全局数据流通过扩展`DataFlow::Configuration`,通过重写ussource和issink谓词,分别限制source入口和sink点

- `isSource` — 定义数据可能从哪里流出
- `isSink` ——定义数据流向何处
- `isBarrier` ——可选，限制数据流
- `isAdditionalFlowStep` — 可选，添加额外的流程步骤

```
/**
 * @kind path-problem
 */

import java
import semmle.code.java.dataflow.DataFlow
import DataFlow::PathGraph


// class constructorref extends RefType{
//     constructorref(){
//         this="test"
//     }
// }


class MyDataFlow extends DataFlow::Configuration{


    MyDataFlow(){
        this="flow"
    }

    override predicate isSource(DataFlow::Node source){

        exists(MethodAccess ac,Method m |m=ac.getCaller() and
m.getParameterType(0).toString()="Map<String,String>" and
source.asExpr()=ac.getArgument(0))


    }


    override predicate isSink(DataFlow::Node sink){
        exists(MethodAccess ac,Method m|ac.getCaller()=m and
m.getDeclaringType().hasQualifiedName("com.blog.devlop.service.impl",
"AccountServiceImpl") and m.hasName("findAccount") and
sink.asExpr()=ac.getArgument(0))

    }

}


from Method m,DataFlow::Node source,DataFlow::Node sink,Class c,Constructor
constructor,Call call,Expr src,MethodAccess ac,MyDataFlow flow
```

```
//where
constructor.getDeclaringType().hasQualifiedName("com.blog.devlop.dao.impl",
"AccountDaoImpl") and call.getCallee()=constructor and
DataFlow::localFlow(DataFlow::exprNode(src),
DataFlow::exprNode(call.getArgument(0)))


//source
//where m=ac.getCaller() and
m.getParameterType(0).toString()="Map<String,String>"
//sink

where flow.hasFlow(source, sink)


select source.getLocation(),source,sink,"test"
```

全局污点跟踪

全局污点跟踪通过引入`TaintTracking::Configuration`,重写`isSource` `isSink`来限制source和sink,最后通过`hasFlow`来判断数据流是否能流通

```
class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
    MyTaintTrackingConfiguration() { this = "MyTaintTrackingConfiguration" }

    override predicate isSource(DataFlow::Node source){

        exists(MethodAccess ac,Method m |m=ac.getCaller() and
m.getParameterType(0).toString()="Map<String,String>" and
source.asExpr()=ac.getArgument(0))


    }


    override predicate isSink(DataFlow::Node sink){
        exists(MethodAccess ac,Method m|ac.getCaller()=m and
m.getDeclaringType().hasQualifiedName("com.blog.devlop.service.impl",
"AccountServiceImpl") and m.hasName("findAccount") and
sink.asExpr()=ac.getArgument(0))

    }
  }
```

语法集合

```
Call和Callable
    Callable表示可调用的方法或构造器的集合
    Call表示调用Callable的这个过程(方法调用 构造器调用等)

MethodAccess 表示方法调用
```

案例

查询指定类中所有方法

```
import java

from Class ccc,Method m,Constructor c, Call call,ExprParent ee, Expr e,Stmt
st,TopLevelType tl,NestedClass nc,TypeVariable type,GenericInterface
map,ParameterizedType pt,TypeBound tb,Variable v,Field f,LocalVariableDecl vv

where
c.getDeclaringType().hasQualifiedName("com.blog.devlop.utils", "jwtutil") and
m.getDeclaringType()=c.getDeclaringType()


select m
```

对代码建立数据流

以一套cvescan代码为例,以登录控制器为例,如何使用codeql建立login方法到daoimpl之间的连接

```java
    @RequestMapping(path = {"/login"})
    public String login(@RequestParam Map<String,String> param) {

        JSONObject json=new JSONObject();
        Map<String,Object> usermap=new HashMap<>();
        String jwt_token=null;


        if(!param.containsKey("username")||!param.containsKey("password")){
            json.put("code","-1");
            return json.toJSONString();

        }

        String username=param.get("username");
        String password=param.get("password");

        Account account=accountService.findAccount(username);

        System.out.println(account.toString());
        if(account==null){
            json.put("msg","user not exists");
            return json.toJSONString();
        }

        if(!account.getPassword().equals(password)){
            json.put("msg","login failed");

            return json.toJSONString();
```

```java

@Repository
public class AccountDaoImpl implements AccountDao {


    @Autowired
    AccountMapper accountMapper;

    @Override
    public Account getAccount(String username) {
        if(username==null){
            return null;
        }
        return accountMapper.getAccount(username);
    }

    @Override
    public boolean update_login_time(String last_login_time, String username) {
        if(last_login_time==null || username==null){
            return false;
        }
        return accountMapper.update_time(last_login_time,username);

    }
}

```

```
/**
 * @kind path-problem
 */

import java
import semmle.code.java.dataflow.DataFlow
import DataFlow::PathGraph
import semmle.code.java.dataflow.TaintTracking
```

```
class MyDataFlow extends DataFlow::Configuration{


    MyDataFlow(){
        this="flow"
    }

    override predicate isSource(DataFlow::Node source){
        exists(MethodAccess ac , Method m
|m.getParameterType(0).toString()="Map<String,String>" and
source.asExpr()=ac.getArgument(0))


    }


    override predicate isSink(DataFlow::Node sink){

        exists(MethodAccess ac,Method m| m.hasName("getAccount") and
ac.getCaller()=m and ac.getArgument(0)=sink.asExpr())

    }

}


from MyDataFlow mydata, Method m,DataFlow::Node source,DataFlow::Node sink,Class
c,Constructor constructor,Call call,Expr src,MethodAccess ac

where mydata.hasFlow(source, sink)


select source.getLocation(),source,sink,"flow"
```