Android Lint 是 SDK Tools 16（ADT 16）开始引入的一个代码扫描工具，通过对代码进行静态分析，可以帮助开发者发现代码质量问题和提出一些改进建议。除了检查 Android 项目源码中潜在的错误，对于代码的正确性、安全性、性能、易用性、便利性和国际化方面也会作出检查。

最近在项目开发过程中发现如果要推进一些中间件的对接工作，不给业务方提供一些对应的工具的情况下，很难推动代码更新迭代的过程。所以我打算通过lint静态代码检查工具，将业务端内出现了我们想改进的地方的代码标错，然后根据错误提示，帮助他们修改以及升级业务代码。

## lint配置

对于执行 Lint 操作的相关配置，是定义在 gradle 文件的 lintOptions 中，可定义的选项及其默认值。

android {

lintOptions {

// 设置为 true，则当 Lint 发现错误时停止 Gradle 构建

abortOnError false

// 设置为 true，则当有错误时会显示文件的全路径或绝对路径 (默认情况下为true)

absolutePaths true

// 仅检查指定的问题（根据 id 指定）

check 'NewApi', 'InlinedApi'

// 设置为 true 则检查所有的问题，包括默认不检查问题

checkAllWarnings true

// 设置为 true 后，release 构建都会以 Fatal 的设置来运行 Lint。

// 如果构建时发现了致命（Fatal）的问题，会中止构建（具体由 abortOnError 控制）

checkReleaseBuilds true

// 不检查指定的问题（根据问题 id 指定）

disable 'TypographyFractions','TypographyQuotes'

// 检查指定的问题（根据 id 指定）

enable 'RtlHardcoded','RtlCompat', 'RtlEnabled'

// 在报告中是否返回对应的 Lint 说明

explainIssues true

// 写入报告的路径，默认为构建目录下的 lint-results.html

htmlOutput file("lint-report.html")

// 设置为 true 则会生成一个 HTML 格式的报告

htmlReport true

// 设置为 true 则只报告错误

ignoreWarnings true

// 重新指定 Lint 规则配置文件

lintConfig file("default-lint.xml")

// 设置为 true 则错误报告中不包括源代码的行号

noLines true

// 设置为 true 时 Lint 将不报告分析的进度

quiet true

// 覆盖 Lint 规则的严重程度，例如：

severityOverrides ["MissingTranslation": LintOptions.SEVERITY\_WARNING]

// 设置为 true 则显示一个问题所在的所有地方，而不会截短列表

showAll true

// 配置写入输出结果的位置，格式可以是文件或 stdout

textOutput 'stdout'

// 设置为 true，则生成纯文本报告（默认为 false）

textReport false

// 设置为 true，则会把所有警告视为错误处理

warningsAsErrors true

// 写入检查报告的文件（不指定默认为 lint-results.xml）

xmlOutput file("lint-report.xml")

// 设置为 true 则会生成一个 XML 报告

xmlReport false

// 将指定问题（根据 id 指定）的严重级别（severity）设置为 Fatal

fatal 'NewApi', 'InlineApi'

// 将指定问题（根据 id 指定）的严重级别（severity）设置为 Error

error 'Wakelock', 'TextViewEdits'

// 将指定问题（根据 id 指定）的严重级别（severity）设置为 Warning

warning 'ResourceAsColor'

// 将指定问题（根据 id 指定）的严重级别（severity）设置为 ignore

ignore 'TypographyQuotes'

}

}

lint checks已经更新到v2版本了，所以配置项必须按照如下规则填写。

apply plugin: 'java-library'

dependencies {

implementation fileTree(dir: 'libs', include: ['\*.jar'])

compileOnly 'com.android.tools.lint:lint-api:26.4.2'

compileOnly 'com.android.tools.lint:lint-checks:26.4.2'

}

sourceCompatibility = "1.8"

targetCompatibility = "1.8"

jar {

manifest {

attributes("Lint-Registry-v2": "com.xxx.lint.router.=IssueRegistry")

}

}

**这里面有个很关键的点 attributes  "Lint-Registry-v2" 这个key必须要写死。value就是项目内的IssueRegistry。**

## 如何实现自定义lint规则

Detector类明确的是第一次我们要检查些什么，这个很简单也很明确，我要检查的是哪些类，是代码还是资源文件或者xml，是构造函数还是方法调用。其次第二点才是哪些边界条件是我们认为有问题的地方，这个地方要抛出一个异常出来，给予开发人员提示。

public class RouteDetector extends Detector implements Detector.UastScanner {

private final String WM\_ROUTER\_PACKAGE = "com.sankuai.waimai.router";

private final String WM\_ROUTER\_ANNOTATION = WM\_ROUTER\_PACKAGE + ".annotation.RouterPage";

private final String WM\_ROUTER\_CALL = WM\_ROUTER\_PACKAGE + ".Router";

static final Issue ISSUE = Issue.create(

"router\_annotation\_issue", //唯一 ID

"不允许使用该注解", //简单描述

"全局项目不允许使用该注解 请更换RouterUri", //详细描述

Category.CORRECTNESS, //问题种类（正确性、安全性等）

6, //权重

Severity.ERROR, //问题严重程度（忽略、警告、错误）

new Implementation( //实现，包括处理实例和作用域

RouteDetector.class,

Scope.JAVA\_FILE\_SCOPE));

static final Issue CALL\_ISSUE = ISSUE.create("router\_call\_issue", //唯一 ID

"不要直接引用WM router", //简单描述

"使用项目封装的路由中间件完成跳转", //详细描述

Category.CORRECTNESS, //问题种类（正确性、安全性等）

6, //权重

Severity.ERROR, //问题严重程度（忽略、警告、错误）

new Implementation( //实现，包括处理实例和作用域

RouteDetector.class,

Scope.JAVA\_FILE\_SCOPE));

@Override

public List<Class<? extends UElement>> getApplicableUastTypes() {

List<Class<? extends UElement>> types = new ArrayList<>();

types.add(UAnnotation.class);

types.add(UCallExpression.class);

return types;

}

@Override

public UElementHandler createUastHandler(@NotNull JavaContext context) {

return new UElementHandler() {

@Override

public void visitAnnotation(@NotNull UAnnotation node) {

isAnnotation(node);

}

private void isAnnotation(UAnnotation node) {

String type = node.getQualifiedName();

if (WM\_ROUTER\_ANNOTATION.equals(type)) {

context.report(ISSUE, node, context.getLocation(node),

"该注解不允许使用");

}

}

@Override

public void visitCallExpression(@NotNull UCallExpression node) {

checkIsMethod(node);

checkIsConstructorCall(node);

}

private void checkIsConstructorCall(UCallExpression node) {

if (!UastExpressionUtils.isConstructorCall(node)) {

return;

}

UReferenceExpression classRef = node.getClassReference();

if (classRef != null) {

String className = UastUtils.getQualifiedName(classRef);

String value = "com.xxx.routerprotocol.request.xxxUriRequest";

String uriValue = WM\_ROUTER\_PACKAGE + ".common.DefaultUriRequest";

String pageValue = WM\_ROUTER\_PACKAGE + ".common.DefaultPageUriRequest";

if (className.equals(value)) {

context.report(CALL\_ISSUE, node, context.getLocation(node),

"请使用项目提供的路由中间件 ");

}

if (className.equals(uriValue) || className.equals(pageValue)) {

context.report(CALL\_ISSUE, node, context.getLocation(node),

"请使用项目提供的路由中间件 ");

}

}

}

private void checkIsMethod(UCallExpression node) {

if (UastExpressionUtils.isMethodCall(node)) {

if (node.getReceiver() != null && node.getMethodName() != null) {

PsiMethod method = node.resolve();

if (context.getEvaluator().isMemberInClass(method, WM\_ROUTER\_CALL)) {

context.report(CALL\_ISSUE, node, context.getLocation(node),

"请使用项目提供的路由中间件");

}

}

}

}

};

}

}

上面是项目lint检查的代码块，主要的核心点在getApplicableUastTypes，你允许type之后才能在下面的UElementHandler被传入截获，这样才能进行代码检查的开发。

UCallExpression 这个类型可以接受代码中的构造器以及方法调用，如果有一些特殊的类或者对象你不允许业务人员使用的情况下你对它进行一个报错处理。

UElementHandler 有个地方一定要注意，你想要做检查的地方一定不要写super，会导致无法继续完成代码。

上述代码完成的功能就是扫描代码是否实现了RouterPager这个注解。然后检测是否项目内引用到了Router这个类。项目内是不是用了过期的UriRequest或者美团的DefaultUriRequest，如果引用到抛出一个issue。

出了上述写的简单的代码扫描功能意外，lint 还能扫描 xml 文件 ，资源文件，gradle文件，通过静态扫描的方式可以更好的帮助我们构建我们的安卓项目。

**下面分析下另外一个 关于资源文件检查的lint**

import com.android.resources.ResourceFolderType;

import com.android.tools.lint.detector.api.\*;

import org.jetbrains.annotations.NotNull;

import java.io.File;

import java.util.EnumSet;

public class PngResourceDetector extends Detector implements Detector.ResourceFolderScanner {

public static final Issue ISSUE = Issue.create(

"image too large",

"Log Usage",

"Please use the unified LogUtil class!",

Category.CORRECTNESS,

6,

Severity.ERROR,

new Implementation(PngResourceDetector.class, Scope.RESOURCE\_FOLDER\_SCOPE)

);

@Override

public boolean appliesToResourceRefs() {

return super.appliesToResourceRefs();

}

@Override

public void checkFolder(@NotNull ResourceContext context, @NotNull String folderName) {

super.checkFolder(context, folderName);

File parent = context.file;

for (File file : parent.listFiles()) {

if (file.isFile()) {

long length = file.length();

System.out.print(file.toString());

if (length > 100) {

context.report(ISSUE, Location.create(file),

"This code mentions `lint`: \*\*Congratulations\*\*");

}

}

}

}

@Override

public boolean appliesTo(@NotNull ResourceFolderType folderType) {

System.out.print(folderType.name());

// return true;

return folderType.compareTo(ResourceFolderType.DRAWABLE) == 0 || folderType.compareTo(ResourceFolderType.MIPMAP) == 0;

}

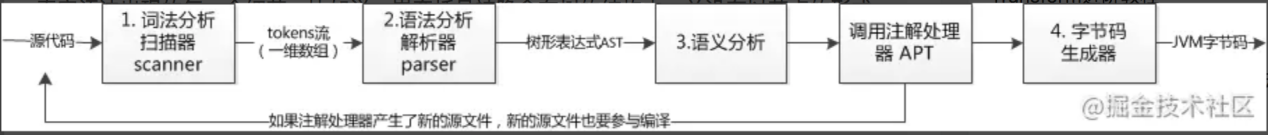
}

你要检查哪些资源你就要实现相关的接口，比如说我要检查java kotlin 代码，那么你要去实现UastScanner，如果你要扫描资源文件，那么你要去实现ResourceFolderScanner。  
之后我们需要复写appliesTo 方法，我们需要扫描哪些资源文件，这个lint只扫描mipmap还有drawable，所以我们只要扫描这两个类型就可以了。  
之后会进入checkFolder 方法内，我们当前做的很简单，检查下文件的length，当文件大小大于多少的时候会提示一个错误。

## AST（Abstract Syntax Tree）抽象语法树

抽象语法树（abstract syntax code，AST）是源代码的抽象语法结构的树状表示，树上的每个节点都表示源代码中的一种结构，这所以说是抽象的，是因为抽象语法树并不会表示出真实语法出现的每一个细节，比如说，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现。抽象语法树并不依赖于源语言的语法，也就是说语法分析阶段所采用的上下文无文文法，因为在写文法时，经常会对文法进行等价的转换（消除左递归，回溯，二义性等），这样会给文法分析引入一些多余的成分，对后续阶段造成不利影响，甚至会使合个阶段变得混乱。因些，很多编译器经常要独立地构造语法分析树，为前端，后端建立一个清晰的接口。

我们简单的从这张图来看下java的AST的过程。



### 步骤一：词法分析，将源代码的字符流转变为 Token 列表。

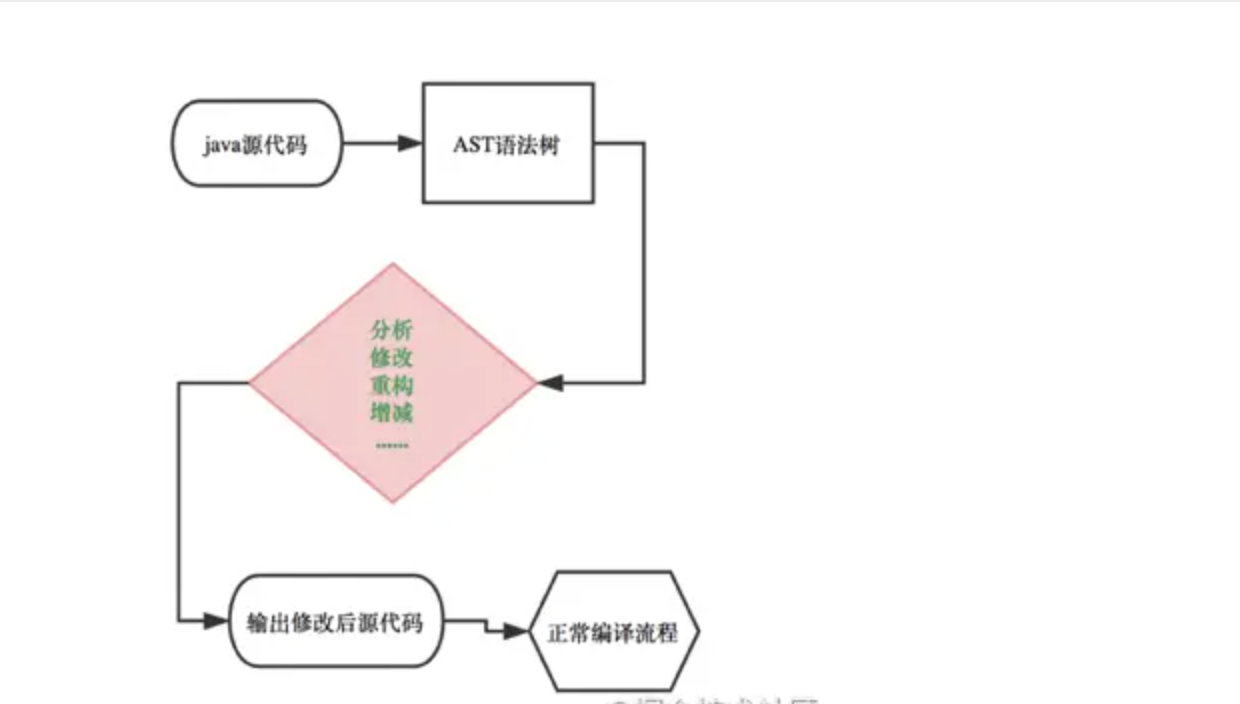
一个个读取源代码，按照预定规则合并成 Token，Token 是编译过程的最小元素，关键字、变量名、字面量、运算符等都可以成为 Token。

### 步骤二：语法分析，根据 Token 流来构造树形表达式也就是 AST。

语法树的每一个节点都代表着程序代码中的一个语法结构，如类型、修饰符、运算符等。经过这个步骤后，编译器就基本不会再对源码文件进行操作了，后续的操作都建立在抽象语法树之上。

apt的过程就是在源代码被转化成ast之后执行的对注解的一次process，所以我们能在apt的过程中获取到所有注解类以及注解的类信息相关。

## Android Lint



而对于Android Lint来说，它本质上就是AST抽象语法树，通过语法树获取到所有代码的节点，之后对其进行自定义的逻辑判断，举个例子，当前类是不是符合了特定标准，比如是不是一个构造器，是不是一个方法，方法名是什么之类的，当符合特定规则之后就会抛出一个Issue。

在Android Lint迭代过程中，扫描源代码的Scanner先后经历了三个版本的AST。

1. 最开始使用的是JavaScanner，Lint通过Lombok库将Java源码解析成AST(抽象语法树)，然后由JavaScanner扫描。
2. 在Android Studio 2.2和lint-api 25.2.0版本中，Lint工具将Lombok AST替换为PSI，同时弃用JavaScanner，推荐使用JavaPsiScanner。

PSI是JetBrains在IDEA中解析Java源码生成语法树后提供的API。相比之前的Lombok AST，可以支持Java 1.8、类型解析等。使用JavaPsiScanner实现的自定义Lint规则，可以被加载到Android Studio 2.2+版本中，在编写Android代码时实时执行。

1. 在Android Studio 3.0和lint-api 25.4.0版本中，Lint工具将PSI替换为UAST，同时推荐使用新的UastScanner。

### UAST

UAST是JetBrains在IDEA新版本中用于替换PSI的API。UAST更加语言无关，除了支持Java，还可以支持Kotlin。

## 在不熟悉API的情况下如何更好的写一个Lint呢？

以我个人的开发经验来看，我会从Android原生提供的Lint规则中去寻找可能适合我的逻辑。举个例子，我之前在使用埋点的时候我不小心给字符串前面加了个空格，我这个时候就会反思，是不是可以通过静态扫描的方式去搞，但是这个时候api不熟悉怎么办呢？？

其实我之前在用TextView的时候发现当我直接设置一个字符串进去的时候lint会爆黄。有思路就可以抄代码，我去找到了SetTextDetector,然后我就根据其中的代码，完成了这个静态扫描工具的开发。

public class EventSpaceDetector extends Detector implements Detector.UastScanner {

static final Issue ISSUE = Issue.create(

"event\_space\_issue", //唯一 ID

"埋点不允许出现空格", //简单描述

"你不知道有时候卵用空格会出问题的吗", //详细描述

Category.CORRECTNESS, //问题种类（正确性、安全性等）

6, //权重

Severity.WARNING, //问题严重程度（忽略、警告、错误）

new Implementation( //实现，包括处理实例和作用域

EventSpaceDetector.class,

Scope.JAVA\_FILE\_SCOPE));

private final String packageName = "com.kronos.sample";

@Override

public List<Class<? extends UElement>> getApplicableUastTypes() {

List<Class<? extends UElement>> types = new ArrayList<>();

types.add(UCallExpression.class);

return types;

}

@Override

public UElementHandler createUastHandler(@NotNull JavaContext context) {

return new UElementHandler() {

@Override

public void visitCallExpression(@NotNull UCallExpression node) {

checkIsConstructorCall(node);

}

private void checkIsConstructorCall(UCallExpression node) {

if (!UastExpressionUtils.isConstructorCall(node)) {

return;

}

UReferenceExpression classRef = node.getClassReference();

if (classRef != null) {

String className = UastUtils.getQualifiedName(classRef);

String value = packageName + ".Event";

List<UExpression> args = node.getValueArguments();

for (UExpression element : args) {

if (element instanceof ULiteralExpression) {

Object stringValue = ((ULiteralExpression) element).getValue();

if (stringValue instanceof String && stringValue.toString().contains(" ")) {

if (!TextUtils.isEmpty(value) && className.equals(value)) {

context.report(ISSUE, node, context.getLocation(node),

"谁给你的胆子用空格的");

}

}

}

element.getExpressionType();

}

}

}

};

}

}

但是实际上我在SetTextDetector中找到了ULiteralExpression,这个就是当前的语法树中的变量值，我将它的value取出来之后，判断了下内容是否含有空格，如果有则在当前地方直接抛出一个issue。

Tree Api + ClassScanner = 识别三方隐私权限调用

我们的主要扫描目标是第三方aar，jar还有class产物。这些其实都是.class文件，因为已经跳过了javac的过程，所以也就没有语法树了。

class PrivacyClassDetector : Detector(), Detector.ClassScanner {

override fun checkClass(context: ClassContext, classNode: ClassNode) {

super.checkClass(context, classNode)

}

override fun checkCall(

context: ClassContext,

classNode: ClassNode,

method: MethodNode,

call: MethodInsnNode

) {

super.checkCall(context, classNode, method, call)

}

override fun getApplicableAsmNodeTypes(): IntArray? {

return intArrayOf(AbstractInsnNode.METHOD\_INSN)

}

override fun checkInstruction(

context: ClassContext,

classNode: ClassNode,

method: MethodNode,

instruction: AbstractInsnNode

) {

super.checkInstruction(context, classNode, method, instruction)

if (instruction is MethodInsnNode) {

if (instruction.isPrivacy() != null) {

print("checkInstruction AbstractInsnNode:${instruction.opcode} \r\n")

context.report(

ISSUE, context.getLocation(instruction),

"外圈q+内圈"

)

}

}

}

private fun MethodInsnNode.isPrivacy(): PrivacyAsmEntity? {

val pair = PrivacyHelper.privacyList.firstOrNull {

val first = it

first.owner == owner && first.code == opcode && first.name == name && first.desc == desc

}

return pair

}

companion object {

val ISSUE = Issue.create(

"ClassSampleDetector", //唯一 ID

"咦 ", //简单描述

"我只是想让你报错而已", //详细描述

Category.CORRECTNESS, //问题种类（正确性、安全性等）

6, Severity.ERROR, //问题严重程度（忽略、警告、错误）

Implementation( //实现，包括处理实例和作用域

PrivacyClassDetector::class.java,

Scope.CLASS\_FILE\_SCOPE

)

)

}

}

interface ClassScanner : FileScanner {

// 需要检查的 node 类型

fun getApplicableAsmNodeTypes(): IntArray?

// 函数调用

fun checkInstruction(

context: ClassContext,

classNode: ClassNode,

method: MethodNode,

instruction: AbstractInsnNode

)

// 和其他lint api 提供的类似

fun getApplicableCallNames(): List<String>?

// 和其他lint api 提供的类似

fun getApplicableCallOwners(): List<String>?

// 和getApplicableCallNames匹配

fun checkCall(

context: ClassContext,

classNode: ClassNode,

method: MethodNode,

call: MethodInsnNode

)

// 和getApplicableCallOwners匹配

fun checkClass(context: ClassContext, classNode: ClassNode)

}

我们先从这个抽象接口看起，我在函数上面做了简单的方法描述。这次隐私api的判断因为我们要判断的栈帧方法比较多，比如INVOKEVIRTUAL,INVOKESTATIC这种都有，所以getApplicableAsmNodeTypes这个上面，我们获取了所有的函数调用。其中INVOKEVIRTUAL,INVOKESTATIC 都是MethodInsnNode，所以我们只要在getApplicableAsmNodeTypes方法中添加AbstractInsnNode.METHOD\_INSN就行了。

然后我们需要做的也很简单，因为我们的输入类型只有MethodInsnNode，所以当checkInstruction就是栈帧调用方法被执行的时候，将call直接转化成MethodInsnNode，之后判断当前栈判断当前方法是不是操作符，描述符，所有者等都符合我们的隐私api的定义，如果是则调用lint repot就行了。

动态可配置化Lint

首先我们看下这部分简单的json定义，因为我们要根据这些json来去做动态化的json匹配。

因为构造函数和方法调用其实是两种不同的lint写法，所以我们在这里定义了两个不同。

{

"methods": [

{

"name\_regex": "android.net.wifi.WifiManager.getSSID",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getWifiName 替换哦",

"excludes": [

"com.xxxxxx.privacy.PrivacyImp"

]

},

{

"name\_regex": "android.net.wifi.WifiManager.getBSSID",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getWifiName 替换哦",

"excludes": [

"com.xxxxx.privacy.PrivacyImp"

]

},

{

"name\_regex": "Settings.Secure.getString",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getAndroidId 替换哦",

"excludes": [

"com.bilibili.privacy.PrivacyImp",

"com.bilibili.adcommon.util.LocationUtil"

]

},

{

"name\_regex": "android.telephony.TelephonyManager.getImei",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getDeviceId 替换哦",

"excludes": [

"com.xxxx.privacy.PrivacyImp"

]

},

{

"name\_regex": "android.telephony.TelephonyManager.getDeviceId",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getDeviceId 替换哦",

"excludes": [

"com.xxxx.privacy.PrivacyImp"

]

},

{

"name\_regex": "android.telephony.TelephonyManager.getDeviceId",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getDeviceId 替换哦",

"excludes": [

"com.xxxx.privacy.PrivacyImp"

]

},

{

"name\_regex": "java.net.getHostAddress",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getIpAddress 替换哦",

"excludes": [

"com.xxxxx.privacy.PrivacyImp"

]

},

{

"name\_regex": "android.content.pm.PackageManager.getInstalledApplications",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getPackageList 替换哦",

"excludes": [

"com.Xxxxxx.privacy.PrivacyImp"

]

},

{

"name\_regex": "android.content.pm.PackageManager.getInstalledPackages",

"message": "gie gie 这个是隐私api 请使用PrivacyUtil.getAppList 替换哦",

"excludes": [

"com.xxxxx.privacy.PrivacyImp"

]

}

],

"constructions": [

{

"name\_regex": "",

"message": ""

}

]

}

以上是我们当前整理的隐私相关的json名单，剔除的则是我们中间件相关的代码，我们希望一次性能将所有隐私相关的api整改到中间件上去。

因为这次诉求比较简单，我们只定义了方法和构造函数两个数组。name\_regex 代表规则匹配，message则标示的是提示文案，excludes代表的是白名单列表。因为我们的诉求其实是统一调用我们定义的中间件，素有中间件都在我们的白名单列表上。

这个地方的难点就在于如何让lint代码能读取到我们的配置的json文件。

class DynamicLint : Detector(), Detector.UastScanner {

lateinit var globalConfig: DynamicConfigEntity

override fun beforeCheckRootProject(context: Context) {

super.beforeCheckRootProject(context)

globalConfig = GsonUtils.inflate(context.project.dir)

}

}

Detector提供了一个beforeCheckRootProject方法。这个方法会把当的目录信息之类的传入，我们就是要通过这个Context上下文，去获取我们的可配置化的json文件信息。

另外这里有个小细节，因为我们的项目采取的是compose building的模式，而这个Context正常传入的只有Module路径，所以这里要进行一个简单的递归查找。

private fun findCodeQuality(projectDir: File): File? {

if (projectDir.parent != null) {

val parent = projectDir.parentFile

val file = parent.listFiles()?.firstOrNull {

it.name == ".codequality" && it.isDirectory

}

return file ?: findCodeQuality(parent)

}

return null

}

一个简单的递归调用寻址。

/\*\*

\* name是完全匹配，nameRegex是正则匹配，匹配优先级上name > nameRegex

\* inClassName是当前需要匹配的方法所在类

\* exclude是要排除匹配的类（目前以类的粒度去排除）

\*/

private fun match(

nameRegex: String?,

qualifiedName: String?,

inClassName: String? = null,

exclude: List<String> = emptyList(),

excludeRegex: String? = null

): Boolean {

qualifiedName ?: return false

//排除

if (inClassName != null && inClassName.isNotEmpty()) {

if (exclude.contains(inClassName)) return false

if (excludeRegex != null &&

excludeRegex.isNotEmpty() &&

Pattern.compile(excludeRegex).matcher(inClassName).find()

) {

return false

}

}

if (nameRegex != null && nameRegex.isNotEmpty() &&

Pattern.compile(nameRegex).matcher(qualifiedName).find()

) {//在匹配nameRegex

return true

}

return false

}

}

代码检查匹配就是通过上述代码进行的.

虽然我们通过lint将项目内的代码进行了一次约束，但是已经编译成.class并不会被这段UastScanner所识别。

可以通过ClassScanner进行class的lint扫描.

而且我们如果将这个需求提测的情况下，对于测试同学来说，这个需求也就没有办法测试了。所以我们需要另外一种方式能在运行时提供一部分hook能力，当这些隐私api被调用的情况下，或是产生一条文件记录或者是直接崩溃都行。

基于Epic动态hook

epic的hook机制，基于art的elft文件格式，所以可以hook所有代码内的方法调用，虽然是被动了点，但是可以避免反射等极端情况导致的隐私权限调用问题，以及第三方sdk内的调用情况。

首先我们可以沿用之前项目内定义好的那份动态json文件,之后通过软连接的方式直接复制到debug的assets文件夹下面。

软连接是linux中一个常用命令，它的功能是为某一个文件在另外一个位置建立一个同不的链接。 具体用法是：ln -s 源文件 目标文件。

fun hookManager(context: Context) {

val steam = context.resources.assets.open("dynamic.json")

val configEntity = GsonUtils.inflate(steam)

configEntity.methods.forEach {

start(it)

}

}

fun start(entity: DynamicEntity) {

if (entity.name\_regex.isNotEmpty()) {

val methodName = entity.name\_regex.substring(entity.name\_regex.lastIndexOf(".") + 1)

val className = entity.name\_regex.substring(0, entity.name\_regex.lastIndexOf("."))

val lintClass = Class.forName(className)

DexposedBridge.hookAllMethods(lintClass, methodName, object : XC\_MethodHook() {

override fun beforeHookedMethod(param: MethodHookParam?) {

super.beforeHookedMethod(param)

Log.i("EpicHook", "EpicHook")

}

})

}

}

之后在debug包的情况下，通过反序列化json，同样生成好对应的hook文件配置,之后调用DexposedBridge.hookAllMethods方法。

虽然我们已经有了动态Hook的能力，但是因为动态hook一定要等到方法被调用的情况下才会执行异常，对于一些调用逻辑比较深的页面，可能会出现覆盖不到的情况。

而更好的方案就是通过asm，去对第三方的隐私代码进行替换，将他们转包到我们的中间件内。

这样就能做到多重保险，可以极大程度的应对机构的审查。

通过Transform + Asm 定位敏感权限

针对deviceid，可以这样做：

通过Asm的Tree api，之后判断当前的方法栈帧是不是"android/telephony/TelephonyManager"的getDeviceId方法，如果是则对其进行修改，替换成我们的定义的静态方法。

这里有个小tips，因为之前必然是获取了android/telephony/TelephonyManager并完成了方法的压栈，所以我们要把上一个方法调用移除。