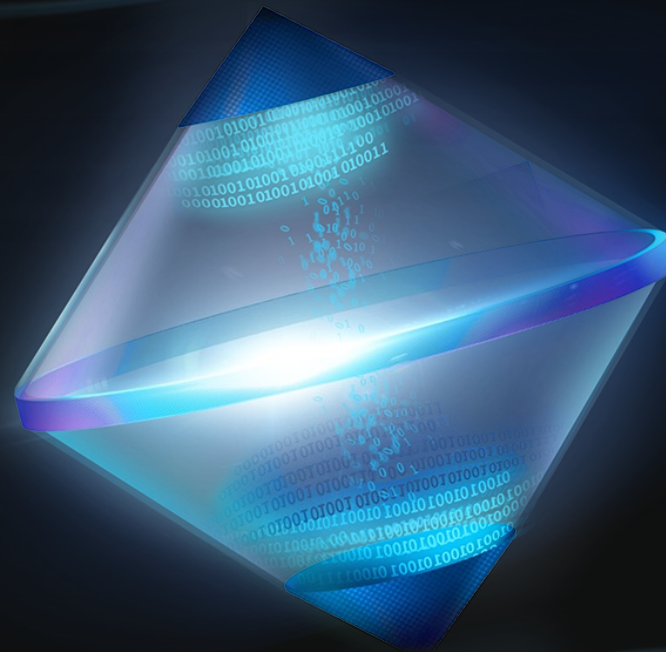


# 闭源系统下的漏洞 自动化发现之旅

演讲人：秦策 (Hearmen)



## ◆ 关于我



望潮实验室（南京）负责人

专注移动安全、应用安全、二进制

BlackHat、Defcon、HITB 等演讲者

KCON2015 演讲者



# 目录

01

**EvilParcel**

系统的银翼杀手

02

**CodeQL**

打破源码限制

03

**LLM**

模型助力漏洞挖掘

04

**总结与展望**

# ◆ EvilParcel

Evilparcel 漏洞是一类存在于 Android 系统中的漏洞，它们利用了 Parcelable 子类在序列化和反序列化过程中实现不匹配的问题

2014

## 首次发现

Michalbednarski 首次发现此类漏洞并交给谷歌官方修复

2017

## 漏洞利用公开

Michalbednarski 公开漏洞利用方法，研究者开始陆续跟进，各大厂商的此类漏洞也开始出现

2022

## Android13 修复

推出 LazyValue 机制，Bundle 中的每个 item 都会有单独的 length

2023

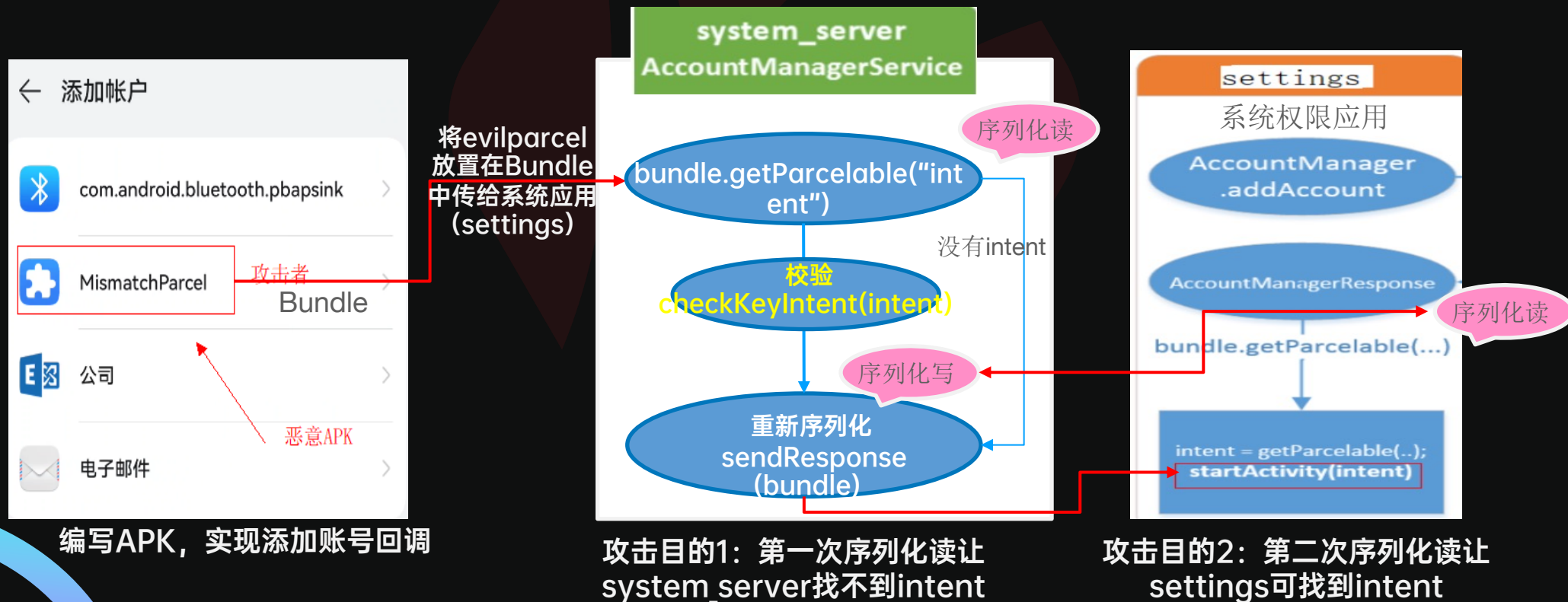
## 安卓系统的碎片化

各个手机厂商自己的 ROM 并不能及时更新  
安卓官方依然接收此类漏洞

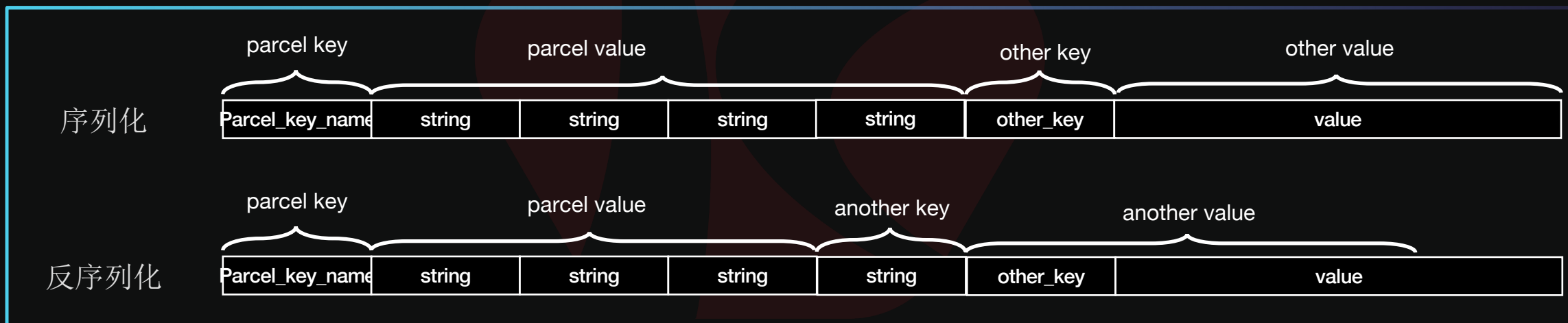
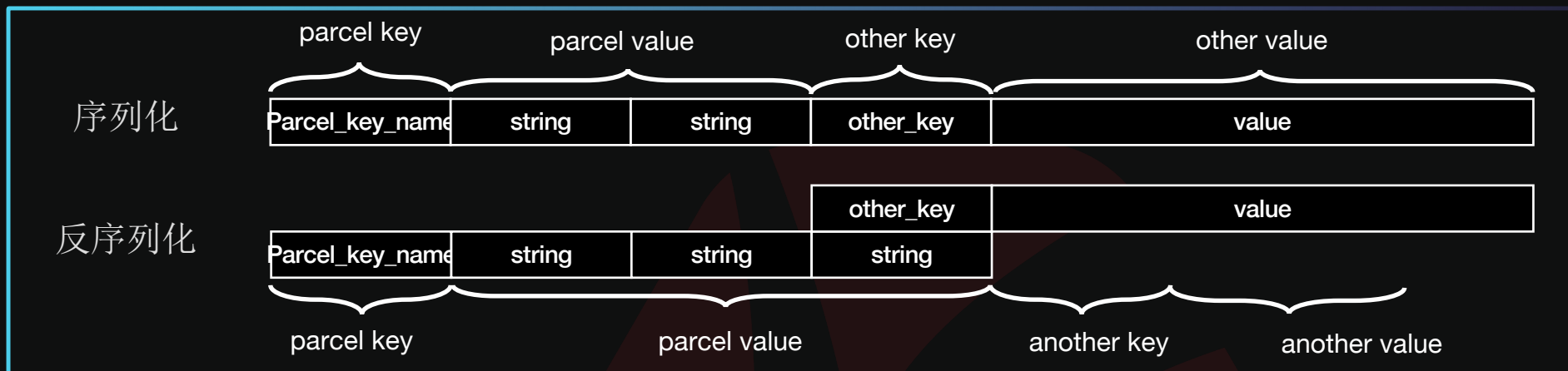
# ◆ EvilParcel

Evilparcel 漏洞可以导致 Bundle 在多次序列化和反序列化过程中内容发生改变。

结合 LAW 跳板，可以实现权限提升，严重影响系统安全



# ◆ EvilParcel



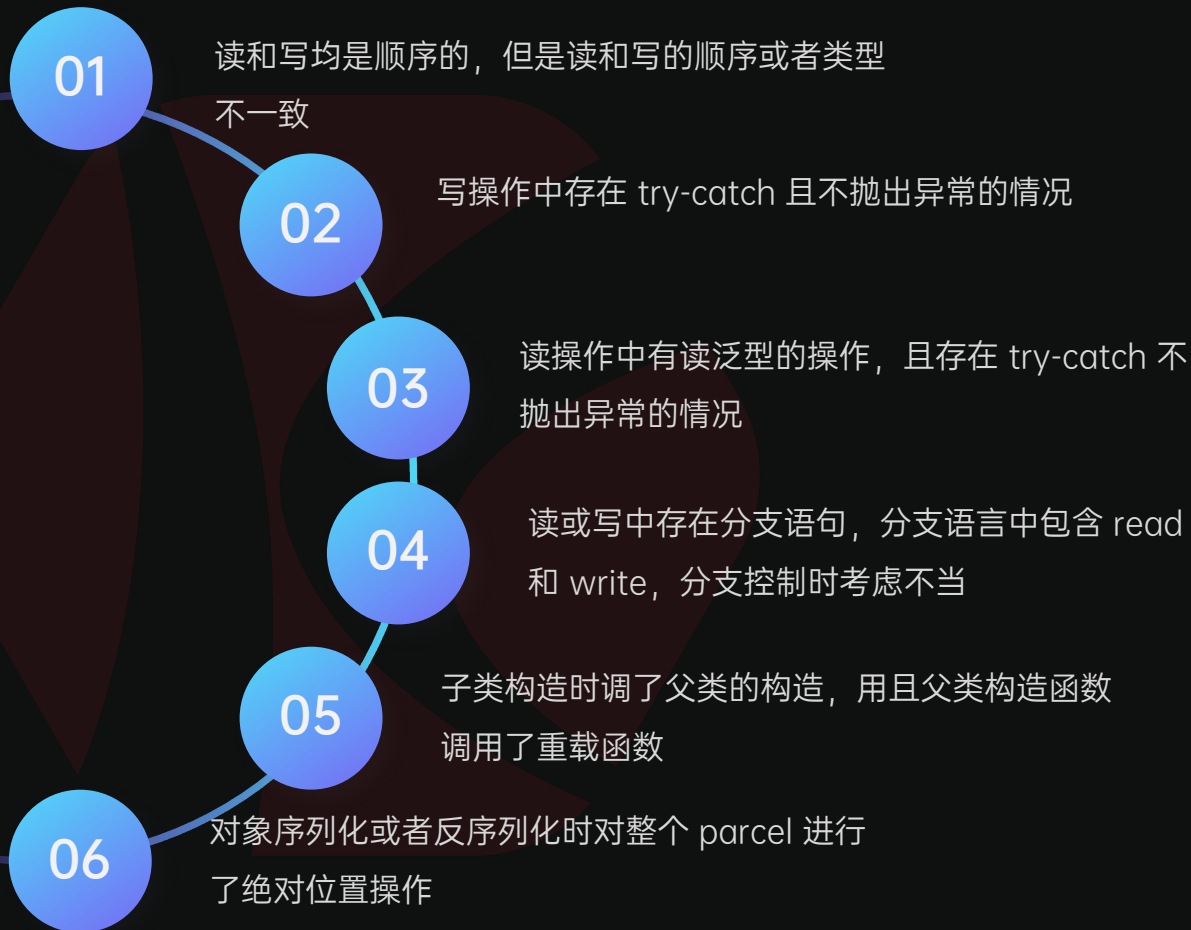
Parcelable 类 `writeToParcel` 函数写入的数据在被 `CREATOR.createFromParcel` 读取时无法一一对应

# ◆ EvilParcel 分类

## 漏洞原语

writeToParcel 函数写入的数据

被 createFromParcel 读取时不匹配



# ◆ EvilParcel Type#1

读和写均是顺序的，但是读和写的顺序或者类型不一致

```
public class ModelOne implements Parcelable {
    private int mDataInt;
    @Override
    public void writeToParcel(Parcel parcel, int i) {
        parcel.writeLong(mDataInt);
    }

    public void readFromParcel(Parcel in) {
        mDataInt = in.readInt();
    }
}
```

```
class ModelOne implements Parcelable {
    Integer mRotation;
    Boolean mIsShared;
    protected ModelOne(Parcel in) {
        mRotation = in.readInt();
    }
    public static final Creator<ModelOne> CREATOR = new
    Creator<ModelOne>() {
        @Override
        public Demo createFromParcel(Parcel in) {
            return new ModelOne(in);
        } };
    @Override
    public void writeToParcel(Parcel parcel, int i) {
        parcel.writeInt(mRotation);
        parcel.writeInt(mIsShared?1:0);
    }
}
```



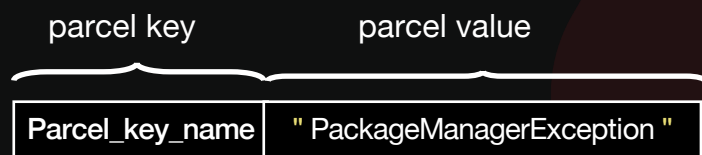
## ◆ EviParcel Type#2

写操作中存在 trycatch 且不抛出异常的情况

```
class ModelTwo extends Parcelable{
    private File mFile;
    public ModelTwo createFromParcel(Parcel source) {
        ModelTwo m = new ModelTwo();
        m.mFile = new File(source.readString());
        return m;
    }
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        try {
            dest.writeString(this.mFile.getCanonicalPath());
        } catch (Exception e) {
            Log.e(TAG, "Exception in writeToParcel: " + e);
        }
    }
}
```

# ◆ EvilParcel Type#3

读操作中有读泛型的操作，且存在 try-catch 捕获异常不抛出



system\_server

PackageManagerException

target app

```
class ModelThree extends Parcelable{
    public ModelThree createFromParcel(Parcel source) {
        try {
            ModelThree m = new ModelThree();
            m.t1 = Class.forName(source.readString());
            m.t2 = source.readInt();
            return m;
        } catch (Exception e) {
            Log.e(TAG, "Exception in createFromParcel: " + e);
            return null;
        }
    }
}
```

## ◆ EvilParcel Type#4

读或写中存在分支语句，且分支语言中包含 read 和 write

需要注意 createByteArray、readByteArray 和 readXXXXList 这种序列化时自动带上长度信息，且未作限制的情况

```
private <T> List<T> readParcelableListInternal() {  
    final int n = readInt();  
    if (n == -1) {  
        list.clear();  
        return list;  
    }  
}  
private <T> List<T> readParcelableListInternal() {  
    int N = val.size();  
    int i=0;  
    writeInt(N);  
}
```

```
class ModelFour extends Parcelable {  
    private Integer mNum;  
    private ArrayList<> mChains;  
    WorkSource(Parcel in) {  
        this.mNum = in.readInt();  
        int numChains = in.readInt();  
        if (numChains > 0) {  
            ArrayList<WorkChain> arrayList = new ArrayList<>(numChains);  
            this.mChains = arrayList;  
            in.readParcelableList(arrayList, WorkChain.class.getClassLoader());  
            return;  
        }  
        this.mChains = null;  
    }  
    public void writeToParcel(Parcel dest, int flags) {  
        dest.writeInt(this.mNum);  
        ArrayList<WorkChain> arrayList = this.mChains;  
        if (arrayList == null) {  
            dest.writeInt(-1);  
            return;  
        }  
        dest.writeInt(arrayList.size());  
        dest.writeParcelableList(this.mChains, flags);  
    }  
}
```

# ◆ EvilParcel Type#5

子类构造时调了父类的构造，用且父类构造函数调用了重写函数

```
class Father implements Parcelable {  
    public Father (Parcel source){  
        readFromParcel(source);  
    }  
  
    public void readFromParcel(Parcel source){  
        this.mEvent = source.readInt();  
    }  
  
    public void writeToParcel(Parcel dest, int flags){  
        dest.writeInt(this.mEvent);  
    }  
}
```

```
class ModelFive extends Father {  
    CREATOR{  
        public Client createFromParcel(Parcel source){  
            return new ModelFive(source);  
        }  
    }  
  
    public ModelFive(Parcel source) {  
        super(source);  
        readFromParcel(source);  
    }  
  
    @Override  
    public void readFromParcel(Parcel source) {  
        this.mProcessName = source.readInt();  
        this.mPkgName = source.readInt();  
    }  
  
    public void writeToParcel(Parcel dest, int flag) {  
        super.writeToParcel(source);  
        dest.writeInt(this.mProcessName);  
        dest.writeInt(this.mPkgName);  
    }  
}
```

Diagram illustrating the call flow between the two classes:

- 1: `ModelFive(Parcel source)` constructor calls `super(source)`.
- 2: `super(source)` calls `readFromParcel(source)` in the `Father` class.
- 3: `readFromParcel(source)` in the `Father` class calls `readFromParcel(source)` in the `ModelFive` class.
- 4: `readFromParcel(source)` in the `ModelFive` class calls `writeToParcel` in the `ModelFive` class.

## ◆ EvilParcel Type#6

对象序列化或者反序列化时对整个 parcel 进行了  
绝对位置操作

```
class ModelSix implements Parcelable {  
    public void readFromParcel(Parcel source) {  
        this.mEvent = source.readInt();  
    }  
    public void writeToParcel(Parcel dest, int flags) {  
        dest.setDataPosition(0);  
        dest.writeInt(this.mEvent);  
    }  
}
```

# ◆ CodeQL

## CodeQL 简介

CODEQL 是 GITHUB 开发的用于自动化安全检测的代码分析引擎。CODEQL 可以根据已有漏洞快速查找相似问题

## CodeQL 使用

CODEQL 的使用可以分为三个步骤：

- 1.根据源码工程创建数据库
- 2.在数据库上运行QL 代码
- 3.人工分析查询结果

## CodeQL 原理

CODEQL 中代码被作为数据进行存储，按照代码的逻辑组织成一个关系型数据库  
数据库中包含有代码的原始信息，抽象语法树信息，数据流信息，控制流信息

## CodeQL 不足

CODEQL 针对编译型语言需要完整的可编译工程  
且由于静态分析的天然缺陷，查询结果会存在漏报和误报



## ◆ CodeQL构建JAVA数据库过程

大多数 ROM 都是闭源的，我们无法获取厂商的源代码  
“For compiled languages, extraction works by monitoring the normal build process.”

```
codeql database create java-database --language=java --  
command='gradle --no-daemon clean test'
```

```
exec "${CODEQL_JAVA_HOME}/bin/java" \ # 执行构建  
    $jvmArgs \  
    --add-modules jdk.unsupported \  
    -cp "${CODEQL_DIST}/tools/codeql.jar" \  
    "com.semmlle.cli2.CodeQL" "$@"
```

# ◆ CodeQL构建JAVA数据库过程

codeql.jar 初始化数据库空间, 生成 log 和 yml 文件,  
并初始化一些环境配置

Init



中间文件和源码打包成codeql数据库  
环境清理

finalize



trace

1. 将 codeql-java-agent.jar 注入到 maven javac 运行环境中
2. hook javac 编译入口获取参数交给 semmlle-extractor-java.jar
3. 正常运行 javac 编译工程, 编译过程不做侵入, 且编译结果不做记录
4. 使用魔改的 javac 编译工程, 忽略编译异常, 并提取过程参数(AST,CFG,DFG)



# ◆ CodeQL构建JAVA数据库过程

构建环境初始化  
构建extractor选择  
javaagent 注入  
日志记录  
构建中间文件打包  
环境清理

codeql.jar

Interceptor 拦截器  
获取 javac 编译参数  
完成 javac 编译流程  
准备 extractor 环境  
调用 extractor

codeql-java-agent.jar

自实现的 java 编译器  
模拟 javac 编译  
抽取编译过程信息  
生成中间数据

semml-extractor-java.jar

**codeql** 并不需要目标工程真正的可编译，只需要获取构建过程中的源码和依赖

# ◆ CodeQL构建JAVA数据库过程

codeql.jar	java.exe	21252	855.77 ...
	conhost.exe	5324	6.28 MB
	tracer.exe	9732	1.2 MB
	conhost.exe	21780	6.31 MB
maven	runner.exe	21488	1.09 MB
	cmd.exe	22520	4.98 MB
	java.exe	9568	64 B/s 0.99 GB
	cmd.exe	18660	4.58 MB
	conhost.exe	24100	6.32 MB
extractor-java.jar	javac.exe	16656	0.08 128 B/s 433.98 ...
	java.exe	22988	26.05 8.5 MB/s 413.95 ...
	conhost.exe	21076	450 B/s 6.32 MB

```
-Xprefer:source
-source
1.8
-target
1.8
-extdirs
C:\Program Files\Java\jdk1.8.0_311\jre\lib\ext;C:\WINDOWS\Sun\Java\lib\ext
-endorseddirs
C:\Program Files\Java\jdk1.8.0_311\jre\lib\endorsed
-bootclasspath
C:\Program Files\Java\jdk1.8.0_311\jre\lib\resources.jar;.....
"-sourcepath"
"D:/Projects/analyzeCodeql/src/main/java;D:/Projects/analyzeCodeql/target/generated-sources/annotations;"
"D:/Projects/analyzeCodeql/src/main/java/Hearmen/Hello.java"
```

# ◆ 无源码构建Java数据库

## 工具逻辑

### 1. 反编译目标程序

```
#> jadx -d test-src -r app-debug.apk
```

2. 拷贝依赖到 bootclasspath 路径 (android.jar、 core-lambda-stubs.jar)

3. 拆解 codeql 构建

## CodeQL构建过程模拟

### 1. 初始化数据库

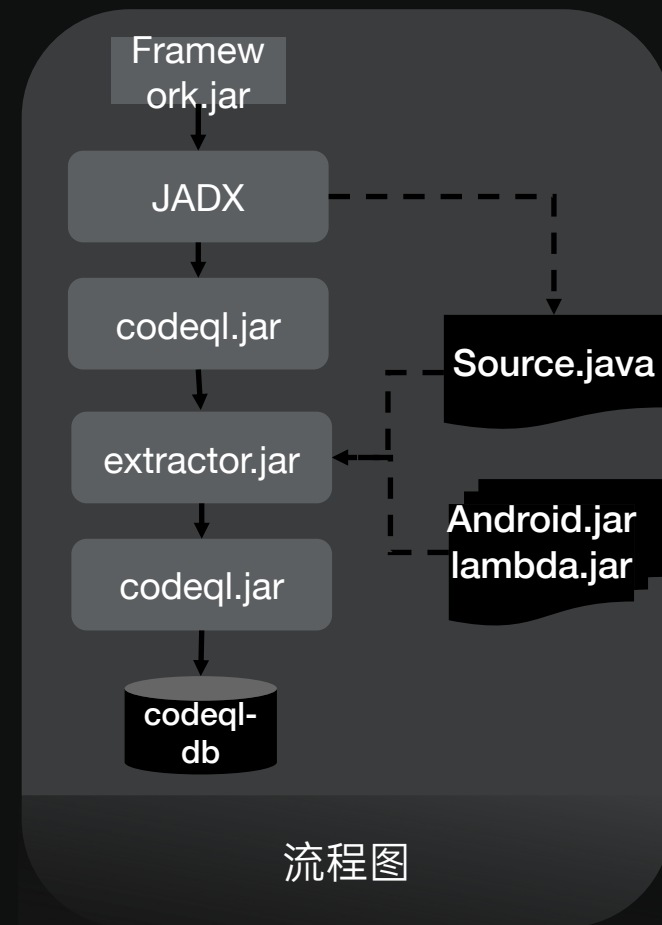
```
#> codeql database init db-name -l java --source-root source
```

2. 调用 extractor 传入编译过程参数 (源码, 依赖)

```
#> java -Xmx1024M -Xms256M -cp semmlle-extractor-java.jar  
com.semmlle.extractor.java.JavaExtractor --javac-args javac.args
```

3. 调用 finalize 生成数据库

```
#> codeql database finalize db-name
```



## ◆ QL 查询语句#1

- 查询所有实现 Parcelable 接口的类
- 保证类在当前的源码库中

```
class ParcelableClass extends Class {  
    ParcelableClass() {  
        this.getAnAncestor*().hasQualifiedName("android.os", "Parcelable")  
    }  
}
```

```
from ParcelableClass type1  
where type1.fromSource()  
select type1
```

# ◆ CodeQL 查询语句 #2

- 查询所有不会抛出异常的 try-catch 语句
- 查询写操作中不抛出异常的 Parcelable 类

```
class NoThrowMethod extends Callable {
    NoThrowMethod() {
        exists(TryStmt try |
            exists(CatchClause ct | try.getACatchClause() = ct )
            and not exists(ThrowStmt te|te.getEnclosingStmt*() =
                try.getACatchClause().getBasicBlock() |
                try.getEnclosingCallable() = this)
        )
    }
}
```

```
class RWParcelableMethod extends Callable {
    RWParcelableMethod() {
        // Parcelable deserialize method. // 用 polyCalls* 会有很多误报
        exists(NoThrowMethod nMethod |
            this.polyCalls(nMethod) or nMethod = this )
    }
}
```

```
class WriteToParcelNoCatchMethod extends RWParcelableMethod {
    WriteToParcelNoCatchMethod() {
        // Parcelable deserialize method.
        (this.getName() = "writeToParcel"
            and this.getDeclaringType().getASourceSupertype*() instanceof
            ParcelableClass)
    }
}
```

# ◆ CodeQL 查询语句

## #3

- 查询所有不会抛出异常的 try-catch 语句
- 编写构造函数中不会抛出异常的 try-catch
- 编写 createFromParcel 中不会抛出异常的 try-catch

- 更好的办法是使用全局污点分析来查找
- 污点分析速度慢，且误报高
- 实际并未发现长调用链情况

```
class CreateFromParcelNoCatchMethod extends RWParcelableMethod {
    CreateFromParcelNoCatchMethod() {
        // Parcelable deserialize method.
        // createFromParcel 一般是 Parcelable 类的内置类的方法
        (this.getName() = "createFromParcel"
         and
         (this.getDeclaringType().getASourceSupertype*()).getEnclosingType()
         instanceof ParcelableClass
         or this.getDeclaringType() instanceof ParcelableClass)
    }
}
```

```
class ConstructorWithParcel extends Constructor {
    ConstructorWithParcel() {
        this.getAParamType().toString().matches("%Parcel")
        and this.getDeclaringType() instanceof ParcelableClass
    }
}
// 查找所有调用到 NoThrowMethod 的构造函数
from ConstructorWithParcel constructor
where exists(NoThrowMethod nMethod |
    constructor.polyCalls(nMethod)
    or nMethod = constructor )
select constructor
```

# ◆ CodeQL 查询语句 #4

- 查询所有读或写中存在分支控制，且分支语言中包含 Parcel.readxxx 和 Parcel.writexxx 的
- 查询类本身没有实现Parcelable 接口但是，在其他 parcelable 类函数中被调用的情况

```
class BranchRWClass extends ParcelableClass {
    BranchRWClass() {
        // Parcelable deserialize method.
        exists(Callable m |
            ((m.getName()= "writeToParcel"
              or m.getName()= "createFromParcel"
              or m instanceof Constructor)
            and m.getDeclaringType() = this)
            or (m.getName()= "createFromParcel"
            and m.getDeclaringType().getASourceSupertype*.getEnclosingType() = this)
            | exists(IfStmt ifs, Call c |
                c.getCallee().getName().regexpMatch("readByteArray|read.*List|create.*Array") |
                ifs.getEnclosingCallable() = m
                and (ifs.getElse() = c.getEnclosingStmt().getBasicBlock()
                    or ifs.getThen() = c.getEnclosingStmt().getBasicBlock())
                )
            or exists(DangerousRead dr | m.polyCalls(dr)) )
        }
}
```

```
class DangerousRead extends Callable {
    DangerousRead(){
        exists(IfStmt ifs, Call c |
            c.getCallee().getName().regexpMatch("readByteArray|read.*List|create.*Array")
            | ifs.getEnclosingCallable() = this and c.getCaller() = this)
        }
}
```

# ◆ CodeQL 查询语句 #5

- 查询所有父类重写函数调用有误的情况
- 查找所有重载函数，此函数在父类构造中被调用，且参数为 Parcel 或者 Parcel 的子类
- 过滤掉一些明显的误报

```
class InheritParcelableMethod extends Callable{
  InheritParcelableMethod(){
    exists(ConstructorWithParcel con|
      con.polyCalls(this)
    ) and
    this.getAnnotation().toString() = "Override"
    and this.getAParamType().toString().matches("%Parcel")
    and not
    this.getName().regexpMatch("onTransact|writeToParcel|createFromParcel|
parcel|unparcel") } }
```



# ◆ CodeQL 查询语句 #6

- 查询所有在 read/write 中直接调用 setDataPosition 进行常量操作的情况

```
class SetDataPositionClass extends ParcelableClass {
    SetDataPositionClass() {
        exists(Callable m |
            ((m.getName()= "writeToParcel"
            or m.getName()= "createFromParcel"
            or m instanceof Constructor)
            and m.getDeclaringType() = this)
            or (m.getName()= "createFromParcel"
            and m.getDeclaringType().getASourceSupertype*().getEnclosingType() = this)
            | exists(MethodAccess ma |
                ma.getCallee().getName()="setDataPosition"
                and ma.getAnArgument().isCompileTimeConstant()
                | ma.getCaller()=m )
            )
        }
    }
}
```

# ◆ CodeQL 查询语句 #7

- JADX 反编译时有可能会失败（出现 goto）
- 需要考虑反编译出错的情况，查询所有反编译失败，且出现在模式调用链中的情况

```
DecompileFailedMethod extends Method {
  DecompileFailedMethod() {
    exists( StringLiteral s |
      s.getValue().matches("Method not decompiled%")
      | s.getAnEnclosingStmt().getEnclosingCallable()=this )
    and exists(Method m |
      m.getName() = "writeToParcel"
      or m.getName() = "createFromParcel"
      | m.polyCalls(this)
      or m=this)
  }
}
```

# ◆ QL 查询结果

evilParcel.ql on framework.jar db - finished in 1 seconds (1081 results) [2023/8/4 11:35:11]

« 1 / 6 »

#select ▾

#	type1
1	Intent
2	ParceledListSlice
3	Region
4	Message
5	KeyEvent
6	AccessibilityEvent
7	AccessibilityNodeInfo
8	AccessibilityWindowInfo
9	AccessibilityServiceInfo
10	GestureStep
11	BaseParceledListSlice
12	ComponentName
13	ResolveInfo
14	ServiceInfo

evilParcel.ql on framework.jar db - finished in 0 seconds (4 results) [2023/8/4 13:47:39]

« 1 / 1 »

#select ▾

#	type3
1	writeToParcel
2	writeToParcel
3	writeToParcel
4	writeToParcel

Quick evaluation counts of evilParcel.ql:247 on framework.jar db - finished in 0 seconds (34 results) [2023/8/4 11:49:18]

« 1 / 1 »

#Quick\_evaluation\_of\_predicate\_Type2 ▾

#	this
1	BaseParceledListSlice
2	createFromParcel
3	Account
4	createFromParcel
5	Intent
6	ParceledListSlice
7	ResolveInfo
8	StringParceledListSlice
9	Configuration
10	createFromParcel
11	NanoAppBinary
12	AudioFormat

evilParcel.ql on framework.jar db - finished in 0 seconds (62 results) [2023/8/4 14:42:34]

« 1 / 1 »

#select ▾

#	type4
1	IntentFilter
2	WifiEnterpriseConfig
3	ApexInfoList
4	PictureInPictureParams
5	PictureInPictureArgs
6	JobInfo
7	WorkSource
8	ProcessStats
9	StackInfo
10	ClientTransaction
11	PackageInfo
12	HistoricalOps
13	Notification
14	NotificationChannel
15	NotificationChannelGroup
16	InstantAppResolveInfo

Quick evaluation counts of evilParcel.ql:183 on framework.jar db - finished in 1 seconds (5 results) [2023/8/4 16:12:11]

« 1 / 1 »

#Quick\_evaluation\_of\_predicate\_InheritParcelableMethod ▾

#	this
1	readParcelableCreator
2	readFromParcel
3	readFromParcel
4	readFromParcel
5	readParcelableCreator

Quick evaluation counts of evilParcel.ql:207 on framework.jar db - finished in 3 seconds (4 results) [2023/8/4 15:26:09]

« 1 / 1 »

#Quick\_evaluation\_of\_predicate\_SetDataPositionCalss2 ▾

#	this
1	SurfaceControl
2	RemoteViews
3	UsageEvents
4	NetworkRegistrationInfo

# ◆ 大语言模型

## 大模型的能力

- 文本生成、文本分类、文本翻译、文本总结、对话问答
- 强大的文本理解能力

## 激发模型的潜能

- 通过提示工程技术，让模型的给出更有质量的回答

## 大模型的局限

- 缺乏获取外部信息能力
- 缺乏实际执行运算能力
- 模型幻觉

## 提升模型的能力

- 在上下文中为模型提供必要的知识
- 为模型提供可调用的工具

# ◆ ChatGPT for EvilParcel

```
@Override
public void writeToParcel(Parcel parcel, int i) {
    parcel.writeLong(mDataInt);
}

public void readFromParcel(Parcel in) {
    mDataInt = in.readInt();
}
}
```

在这段代码中，没有明显的 evilparcel 漏洞。这段代码实现了 Android 中的 Parcelable 接口，用于在不同组件之间传递对象。

"evilparcel" 漏洞通常出现在自定义的 `writeToParcel` 和 `createFromParcel` 方法中，其中对象的字段的读写顺序不匹配，从而导致数据错误或安全问题。在这段代码中，`writeToParcel` 和 `readFromParcel` 方法都使用了 `mDataInt` 这个字段，并且它们的顺序也是匹配的。

因此，根据提供的代码，我没有发现明显的 "evilparcel" 漏洞。

```
create(
```

```
': '你是一个安卓安全专家，我是一个安全研究者，请帮我进行代码审计'}
'请告诉我什么是 EvilParcel 漏洞，并给出示例代码'}
```

```
age']['content'])
```

Python

EvilParcel漏洞是一种安卓应用程序中的代码漏洞，它利用了Android的Binder机制中的一个安全问题。Binder是Android中用于进程间通信的机制，而EvilParcel漏洞允许恶意应用程序通过构造恶意的Parcel对象来执行任意代码。

# ◆ 激发模型潜能

If a machine answer every question, the winner is who asked the best



# ◆ Chain Of Thought

问: 罗杰有 5 个网球。他又买了两盒网球,每盒有 3 个网球。他现在有多少网球?

答: 罗杰一开始有 5 个网球, 2 盒 3 个网球, 一共就是  $2 \times 3 = 6$  个网球。  $5 + 6 = 11$ 。答案是 11

问: 食堂有 23 个苹果, 如果他们吃掉 20 个后又买了 6 个。他们现在有多少个苹果?

模型输入

答: 食堂原来有 23 个苹果, 他们用掉 20 个, 所以还有  $23 - 20 = 3$  个。他们又买了 6 个, 所以现在有  $3 + 6 = 9$  个。答案是 9

模型输出

# ◆ Prompt for EvilParcel

## Prompt

ROLE

规定大模型的角色、场景、任务

CONTEXT

为大模型输入必要的领域知识

EXAMPLE

引入思维链将漏洞的分析过程输入给大模型

QUESTION

规定大模型的任务以及输出格式

CODE

输入要进行检查的全量代码

```
<ROLE>
你是一个安卓安全专家，精通JAVA语言。我是一个安卓安全研究者。现在请你协助我进行安卓的源码审计，发现代码中的安全漏洞。</ROLE>
```

```
<CONTEXT>
EvilParcel 是指存在类 ClassA，其成员函数 writeToParcel 写入的值与内置类 ClassA$CREATOR 的成员函数 createFromParcel 读取的值不匹配。
</CONTEXT>
```

```
<EXAMPLE>
下面我将给出一些实例和解释以帮你更好的理解 EvilParcel
  <EXAMPLE1></EXAMPLE1>
  <EXAMPLE2></EXAMPLE2>
</EXAMPLE>
```

```
<QUESTION>
请一步一步分析下面的代码是否存在 EvilParcel，如果存在，则输出类名，并给出分析过程
</QUESTION>
```

```
<CODE>
  <CODE1>
  </CODE1>
</CODE>
```



# ◆ Prompt for EvilParcel

特殊语法的说明 (try-catch、readXXXList)

writeToParcel

createFromparcel

writeToParcel VS createFromparcel

```
<EXAMPLE1>
```

```
...
```

```
class Demo implements Parcelable {  
    //...  
}
```

```
...
```

writeToParcel: 类 Demo 的成员函数 writeToParcel 写入时写入了4个 Int 类型数据: mRotation、mSurfaceGroupId、mSurfaceType 以及根据 mIsShared 写入的 0 或者 1;  
createFromParcel: 类 Demo 的内置类的成员函数 Demo\$CREATOR.createFromParcel 读取数据时读取了三个 Int 类型: mRotation, mSurfaceGroupId, mSurfaceType;  
会出现读写不匹配。因此上述代码中存在 EvilParcel 类 Demo 的 createFromParcel 和 writeToParcel 相比少读了一个 int 类型

```
</EXAMPLE1>
```

# ◆ Prompt for EvilParcel

2 天

```
type1_checker.call_gpt_check(code)
✓ 4.7s Python

}

public static final Creator<MyMismatchParcel> CREATOR =
new Creator<MyMismatchParcel>() {
    @Override
    public MyMismatchParcel createFromParcel(Parcel in)
    {
        return new MyMismatchParcel(in);
    }

    @Override
    public MyMismatchParcel[] newArray(int size) {
        return new MyMismatchParcel[size];
    }
};

@Override
public int describeContents() {
    return 0;
}

@Override
public void writeToParcel(Parcel parcel, int i) {
    parcel.writeLong(mDataInt);
}

public void readFromParcel(Parcel in) {
    mDataInt = in.readInt();
}
}

'这段代码存在 EvilParcel。 \n\n在类 MyMismatchParcel 中，成员函数 writeToParcel 写入了一个 long 类型的 mDataInt，而成员函数 readFromParcel 读取的是一个 int 类型的 mDataInt。因此，在阅读过程中类型不匹配，存在 EvilParcel。'
```

13 个 EvilParcel

# ◆ 总结和展望

## 为什么是 EvilParcel

- EvilParcel 特点非常明显，且大多情况下仅涉及某一个具体的类而无需分析数据流
- CodeQL 和 LLM 在针对局部问题点的分析上效果非常好

## 技术延伸

- 本议题中的技术不仅可以用在闭源安卓系统中挖掘 EvilParcel，还可以应用到其他 Java 场景的其他漏洞中
- 只需要给定依赖就可以生成数据库，通过定制化的规则提升审计效率
- 可以更进一步，使用 LLM 写 exploit

## 目前的局限

- LLM 辅助漏洞挖掘需要大量前置知识和上下文信息，token 限制 —— LangChain 拆解分析过程
- LLM 也会有幻觉情况出现（误报） —— 迭代 prompt，近义词替换

## ◆ 致谢

感谢 舒超@michael 对工作的支持和信任，以及对技术知识的认可和鼓励

感谢 周全昌 在漏洞学习过程中的指导和在文档撰写过程中的帮助

感谢 招财大土豆@wnagzihxa1n 的分享与思路提供

感谢 杜培 在AI领域的指导

感谢 马卓

# 感谢您的观看!

THANK YOU FOR YOUR WATCHING

