

# INFER: A STATIC ANALISYS TOOL FOR JAVA

## Indice generale

1. INFER.....	2
2. LANG3.....	3
OUTPUT DI INFER.....	3
ANALISI.....	5
RISULTATI.....	8
3. JAVAPARSER.....	9
OUTPUT DI INFER.....	9
ANALISI.....	11
RISULTATI.....	14

# 1. INFER

Infer è un analizzatore statico di programmi per Java, C e Objective-C, scritto in OCaml. Infer è utilizzato all'interno di Facebook e viene eseguito continuamente per verificare alcune proprietà di ogni modifica del codice per le principali app di Facebook per Android e iOS, Facebook Messenger, Instagram e altre app. Può essere utilizzato anche per altri codici: Infer può analizzare anche codice C e codice Java che non sia per Android.

L'analisi statica, chiamata anche analisi statica del codice, è un metodo di debug di un programma per computer che viene eseguito esaminando il codice senza eseguire il programma. Il processo fornisce una comprensione della struttura del codice e può aiutare a garantire che il codice aderisca agli standard del settore. L'analisi statica viene utilizzata nell'ingegneria del software dai team di sviluppo del software e di garanzia della qualità. Il software eseguirà la scansione di tutto il codice in un progetto per verificare la presenza di vulnerabilità durante la convalida del codice.

Infer, in particolare, dà la possibilità di analizzare sia un singolo file, sia interi progetti, costruiti per mezzo di software di build automation come Maven, Make, Ant, Buck, Gradle, consentendo di individuare oltre 60 tipi di vulnerabilità. Tuttavia, non è detto che tutte le vulnerabilità individuate da Infer siano reali: non è raro infatti che alcune di esse risultino dei falsi positivi. È per tale motivo che è necessario validare le vulnerabilità trovate con degli opportuni test.

È a questo scopo che in seguito saranno riportate le analisi di due progetti Java al fine di costruire dei case study per Infer: il progetto di Apache Lang3 e Javaparser. Le vulnerabilità saranno validate creando degli unit test utilizzando JUnit.

## 2. LANG3

Il primo progetto sottoposto ad analisi è Lang3, una libreria open source per Java sviluppata da Apache che permette di implementare numerose funzionalità per rendere più semplice la programmazione, come la manipolazione delle stringhe, la gestione delle date e degli orari, l'interazione con i file e le directory.

La versione presa in esame è quella contenente una vulnerabilità nota, contrassegnata come Lang-703, il cui fix risale al 03/07/2011. È stata recuperata utilizzando Defects4j, un repository accessibile su GitHub che è, per definizione, una collezione di bug riproducibili e un'infrastruttura di supporto con l'obiettivo di far progredire la ricerca in ingegneria del software.

La vulnerabilità presente consiste in due Null Pointer Exception nel metodo Join della classe StringUtils.

## OUTPUT DI INFER

Come primo passo è stato avviato Infer utilizzando il comando *infer run – ant* nella directory del progetto dove è presente il file di configurazione di Ant “build.xml”. L’output viene generato all’interno della directory “infer-out”, creata automaticamente nella stessa directory del progetto target. Il risultato dell’analisi è contenuto nel file “result.txt” ed è il seguente:

#0

*src/main/java/org/apache/commons/lang3/AnnotationUtils.java:74: error: Null Dereference  
object returned by `getAllInterfaces(cls)` could be null and is dereferenced at line 74.*

```
72.     protected String getShortClassName(java.lang.Class<?> cls) {  
73.         Class<? extends Annotation> annotationType = null;  
74. >     for (Class<?> iface : ClassUtils.getAllInterfaces(cls)) {  
75.         if (Annotation.class.isAssignableFrom(iface)) {  
76.             @SuppressWarnings("unchecked")
```

#1

*src/main/java/org/apache/commons/lang3/CharSetUtils.java:194: error: Null Dereference  
object `chars` last assigned on line 189 could be null and is dereferenced at line 194.*

```
192.     int sz = chrs.length;
```

```
193.      for(int i=0; i<sz; i++) {
194. >      if(chars.contains(chrs[i]) == expect) {
195.          buffer.append(chrs[i]);
196.      }
```

#2

src/main/java/org/apache/commons/lang3/builder/ToStringBuilder.java:226: error: Null  
Dereference

object `null` is dereferenced by call to `ToStringBuilder(...)` at line 226.

```
224.      */
225.      public ToStringBuilder(Object object) {
226. >      this(object, null, null);
227.      }
228.
```

#3

src/main/java/org/apache/commons/lang3/concurrent/MultiBackgroundInitializer.java:167:  
warning: Thread Safety Violation

Read/Write race. Non-private method `MultiBackgroundInitializer.getTaskCount()` reads without synchronization from container `this.childInitializers` via call to `Map.values()`. Potentially races with write in method `MultiBackgroundInitializer.addInitializer(...)`.

Reporting because another access to the same memory occurs on a background thread, although this access may not.

```
165.      int result = 1;
166.
167. >      for (BackgroundInitializer<?> bi : childInitializers.values()) {
168.          result += bi.getTaskCount();
169.      }
```

#4

src/main/java/org/apache/commons/lang3/math/NumberUtils.java:518: error: Null Dereference

object `f` last assigned on line 517 could be null and is dereferenced at line 518.

```
516.          try {
```

```

517.          Float f = NumberUtils.createFloat(numeric);
518. >          if (!(f.isInfinite() || (f.floatValue() == 0.0F && !allZeros))) {
519.              //If it's too big for a float or the float value = 0 and the string
520.              //has non-zeros in it, then float does not have the precision we want

```

#5

*src/main/java/org/apache/commons/lang3/math/NumberUtils.java:532: error: Null Dereference  
object `d` last assigned on line 531 could be null and is dereferenced at line 532.*

```

530.          try {
531.              Double d = NumberUtils.createDouble(numeric);
532. >          if (!(d.isInfinite() || (d.floatValue() == 0.0D && !allZeros))) {
533.              return d;
534.          }

```

*Found 6 issues*

*Issue Type(ISSUED\_TYPE\_ID): #*

*Null Dereference(NULL\_DEREFERENCE): 5*

*Thread Safety Violation(THREAD\_SAFETY\_VIOLATION): 1*

Come si può notare, Infer ha individuato in totale 6 vulnerabilità: 5 di tipo Null Dereference, 1 di tipo Thread Safety Violation (in particolare, una possibile Race Condition Read/Write). Non risulta rilevata la vulnerabilità nota.

Di seguito i test per verificare la veridicità delle vulnerabilità rilevate per identificare eventuali falsi positivi.

## ANALISI

**Null Dereference #0:** Infer segnala che l'oggetto ritornato da *getAllInterfaces(cls)* può essere nullo ed è dereferenziato alla riga 74; *getAllInterfaces(cls)* è invocato all'interno del metodo *getShortClassName(java.lang.Class<?> cls)* e ritorna un valore null se *cls* è null. Di seguito il test effettuato.

```

@Test
public void testForNullPointerDereference_0() {
    java.lang.Class<?> cls=null;

    try{
        Class<? extends Annotation> annotationType = null;
        for (Class<?> iface : ClassUtils.getAllInterfaces(cls)) {
            if (Annotation.class.isAssignableFrom(iface)) {
                @SuppressWarnings("unchecked")
                //because we just checked the assignability
                Class<? extends Annotation> found = (Class<? extends Annotation>) iface;
                annotationType = found;
                break;
            }
        }
        String s = new StringBuilder(annotationType == null ? "" : annotationType.getName()).insert(0, '@').toString();
        return;
    }
    catch (NullPointerException e){
        fail();
    }
}

```

Ciò che è stato fatto è l’inizializzazione un oggetto di tipo *java.lang.Class* a null e l’esecuzione delle istruzioni contenute nel metodo *getShortClassName(java.lang.Class<?> cls)*. Il test fallisce se viene generata un’eccezione di tipo *NullPointerException*.

**Null Dereference #1:** Infer segnala che l’oggetto *chars* può essere nullo ed è dereferenziato alla riga 194. Il valore di *chars* viene impostato dal valore di ritorno di *CharSet.getInstance(set)*, che ritorna null se *set* è null. Tuttavia, queste istruzioni fanno parte del corpo del metodo *modify(String str, String[] set, Boolean expect)* che è un metodo privato. I metodi che chiamano il metodo *modify* sono *keep(String str, String[] set)* e *delete(String str, String[] set)*. Di seguito il test effettuato.

```

@Test
public void testForNullPointerDereference_1()
{
    String str = "a"; String[] set = null; //Boolean expect=true;
    try
    {
        String c = CharSetUtils.keep(str, set);
    }
    catch (NullPointerException e)
    {
        fail(message: "Null Pointer Exception 1:keep");
    }

    try
    {
        String c = CharSetUtils.delete(str, set);
    }
    catch (NullPointerException e)
    {
        fail(message: "Null Pointer Exception 1:delete");
    }
}

```

Ciò che è stato fatto è l’inizializzazione dei parametri presi in input dai due metodi in questione, con in particolare *String[] set* inizializzato a null, per verificare se questo tipo di valore viene gestito con le chiamate ai metodi *keep* e *delete*, che a loro volta chiamano al loro interno il metodo *modify*. Il test fallisce se viene generata un’eccezione di tipo *NullPointerException*.

**Null Dereference #2:** Infer individua una possibile Null Dereference quando viene chiamato il costruttore di `ToStringBuilder` con i seguenti parametri `ToStringBuilder(Object obj, null, null)`. Di seguito il test effettuato.

```
@Test
public void testForNullPointerDereference_2()
{
    Integer c=3;
    try{
        ToStringBuilder b=new ToStringBuilder(c, style: null,buffer: null);
    }
    catch(NullPointerException e){
        fail(message: "Null Pointer Exception 2");
    }
}
```

Il test chiama il costruttore di `ToStringBuilder` con i parametri considerati problematici e fallisce se viene generata un'eccezione di tipo `NullPointerException`.

**Thread Safety Violation #3:** Infer riporta che il metodo `getTaskCount(...)` legge senza sincronizzazione da `childInitializer(...)` e che ciò potrebbe causare una race condition di tipo read/write con il metodo `addInitializer(...)`. Verificare con un test la presenza di una race condition non è un'operazione immediata e precisa, tuttavia si può analizzare il codice del software per rendersi conto della plausibilità della vulnerabilità. In seguito riportato il codice preso in analisi.

N.b. le istanze della classe `MultiBackgroundInitializer` sono eseguite in processi paralleli.

```
public void addInitializer(String name, BackgroundInitializer<?> init) {
    if (name == null) {
        throw new IllegalArgumentException(
            "Name of child initializer must not be null!");
    }
    if (init == null) {
        throw new IllegalArgumentException(
            "Child initializer must not be null!");
    }

    synchronized (this) {
        if (isStarted()) {
            throw new IllegalStateException(
                "addInitializer() must not be called after start()!");
        }
        childInitializers.put(name, init);
    }
}
```

```
protected int getTaskCount() {
    int result = 1;

    for (BackgroundInitializer<?> bi : childInitializers.values()) {
        result += bi.getTaskCount();
    }

    return result;
}
```

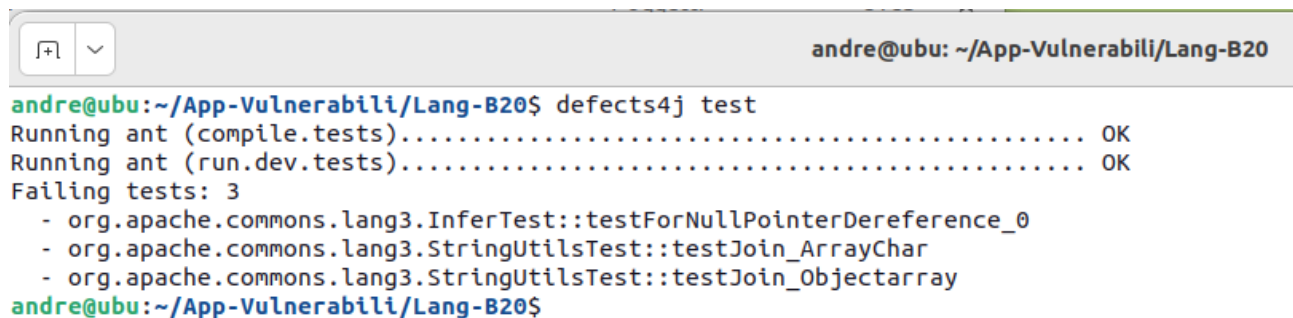
**Null Dereference #4:** Infer segnala che l'oggetto `f` può essere nullo ed è dereferenziato alla linea 518. `f` avrà valore nullo se `NumberUtils.createFloat(numeric)` ritorna un valore nullo e ciò accade quando `numeric` è null. Queste istruzioni fanno parte del corpo del metodo `createNumber(String str)`. L'istruzione presa in esame viene eseguita se l'ultimo carattere di `str` corrisponde al carattere "f" o "F" e `numeric` assume il valore di `str` meno il carattere F. Per come è scritto il metodo quindi

risulta impossibile eseguire l'istruzione in questione se la stringa è nulla. Se *str* = "F" o "f", il valore assunto da *numeric* sarà uguale alla stringa vuota "", che non è null.

**#Null Dereference #5:** Infer segnala che l'oggetto *d* può essere nullo ed è dereferenziato alla linea 532. Valgono le considerazioni fatte per la vulnerabilità #4.

## RISULTATI

Di seguito i risultati dei test eseguiti:



```
andre@ubu: ~/App-Vulnerabili/Lang-B20$ defects4j test
Running ant (compile.tests)..... OK
Running ant (run.dev.tests)..... OK
Failing tests: 3
- org.apache.commons.lang3.InferTest::testForNullPointerDereference_0
- org.apache.commons.lang3.StringUtilsTest::testJoin_ArrayChar
- org.apache.commons.lang3.StringUtilsTest::testJoin_Objectarray
andre@ubu:~/App-Vulnerabili/Lang-B20$
```

Come è possibile notare, l'unico test che fallisce è il test relativo alla vulnerabilità #0. Sembrerebbe quindi che solo questa vulnerabilità sia reale, ma se si allarga lo sguardo oltre l'implementazione del metodo che la contiene e si considerano anche i punti del codice dove viene invocato, si può notare che invece non viene mai invocato. Essendo un metodo protected, in queste condizioni non c'è modo di accedervi. Infatti, in questa versione, Lang3 invoca un override di questo metodo.

Dunque è un falso positivo? La risposta è sia sì che no. Lo è in questa versione della libreria, ma potrebbe non esserlo in futuro nel caso in cui in una versione successiva gli sviluppatori decidessero di utilizzare di nuovo questo metodo e non il suo override. In questo caso la vulnerabilità dovrà essere gestita.

Anche la vulnerabilità #3 è possibile sia reale. Analizzando il codice infatti si può notare la mancanza di sincronizzazione tra i metodi *getTaskCount(...)* e *addInitializer(...)*, cioè tra due metodi che accedono alla stessa area di memoria, uno in lettura e l'altro in scrittura, eseguiti su processi paralleli. Si può quindi affermare che effettivamente questa vulnerabilità è reale.



### 3. JAVAPARSER

Il secondo progetto Java preso in analisi è Javaparser, una delle più popolari librerie che offrono molte funzioni per il parsing di codice java, con cui si può analizzare, manipolare e generare codice sorgente. La versione analizzata è l'ultima disponibile nel momento in cui è stata scritta la medesima relazione, ovvero la versione 3.25.0.

### OUTPUT DI INFER

L'analisi effettuata da infer segnala la presenza di 39 vulnerabilità, di cui 32 di tipo "Null Dereference", 4 di tipo "Inefficient Keyset Operator" e 3 di tipo "Resource leak". Al fine di creare un case study significativo, non è necessario validare tutte le 39 vulnerabilità: si procederà validando alcune vulnerabilità per tipo.

Di seguito l'output di Infer relativo alle vulnerabilità estratte dalle 39 segnalate:

#1

*javaparser-core-serialization/src/main/java/com/github/javaparser/serialization/  
JavaParserJsonDeserializer.java:89: warning: Inefficient Keyset Iterator*

*Accessing a value using a key that was retrieved from a `keySet` iterator. It is more efficient to use an iterator on the `entrySet` of the map, avoiding the extra `HashMap.get(key)` lookup.*

```
87.                .findFirst();
88.                if (!optionalPropertyMetaModel.isPresent()) {
89. >                deferredJsonValues.put(name, nodeJson.get(name));
90.                continue;
91.            }
```

#2

*javaparser-core-serialization/src/main/java/com/github/javaparser/serialization/  
JavaParserJsonDeserializer.java:121: warning: Inefficient Keyset Iterator*

*Accessing a value using a key that was retrieved from a `keySet` iterator. It is more efficient to use an iterator on the `entrySet` of the map, avoiding the extra `HashMap.get(key)` lookup.*

```
119.
120.                for (String name : deferredJsonValues.keySet()) {
121. >                if (!readNonMetaProperties(name, deferredJsonValues.get(name), node)) {
```

```
122.         throw new IllegalStateException("Unknown propertyKey: " +
nodeMetaModel.getQualifiedClassName() + "." + name);
123.     }
```

#4

*javaparser-core/src/main/java/com/github/javaparser/ast/expr/CharLiteralExpr.java:99: error: Null Dereference*

*object returned by `unescapeJava(LiteralStringValueExpr.value)` could be null and is dereferenced at line 99.*

```
97.     */
98.     public char asChar() {
99. >         return StringEscapeUtils.unescapeJava(value).charAt(0);
100.    }
101.
```

#5

*javaparser-core/src/main/java/com/github/javaparser/ast/visitor/CloneVisitor.java:63: error: Null Dereference*

*object `annotations` last assigned on line 60 could be null and is dereferenced by call to `PackageDeclaration(...)` at line 63.*

```
61.     Name name = cloneNode(n.getName(), arg);
62.     Comment comment = cloneNode(n.getComment(), arg);
63. >     PackageDeclaration r = new PackageDeclaration(n.getTokenRange().orElse(null),
annotations, name);
64.     r.setComment(comment);
65. n.getOrphanComments().stream().map(Comment::clone).forEach(r::addOrphanComment);
```

#37

*javaparser-symbol-solver-core/src/main/java/com/github/javaparser/symbolsolver/resolution/typesolvers/AarTypeSolver.java:56: error: Resource Leak*

*resource of type `java.util.jar.JarFile` acquired by call to `JarFile(...)` at line 53 is not released after line 56.*

```
54.     ZipEntry classesJarEntry = jarFile.getEntry("classes.jar");
55.     if (classesJarEntry == null) {
```

```
56. >      throw new IllegalArgumentException(String.format("The given file (%s) is
malformed: entry classes.jar was not found", aarFile.getAbsolutePath()));
57.      }
58.      delegate = new JarTypeSolver(jarFile.getInputStream(classesJarEntry));
```

#38

*javaparser-symbol-solver-core/src/main/java/com/github/javaparser/symbolsolver/resolution/*  
*typesolvers/AarTypeSolver.java:58: error: Resource Leak*

*resource of type `java.util.jar.JarFile` acquired by call to `JarFile(...)` at line 53 is not released*  
*after line 58.*

```
56.      throw new IllegalArgumentException(String.format("The given file (%s) is
malformed: entry classes.jar was not found", aarFile.getAbsolutePath()));
57.      }
58. >      delegate = new JarTypeSolver(jarFile.getInputStream(classesJarEntry));
59.      }
60.
```

## ANALISI

**Inefficient Keyset Iterator #1:** Infer segnala che è stato utilizzato l'iteratore `KeySet()`, ma che sarebbe più efficiente utilizzare il metodo `EntrySet()`, per evitare la chiamata extra `HashMap.get(key)` per ottenere il valore associato alla chiave. Il metodo `EntrySet()` infatti ritorna un oggetto di tipo `EntrySet` i cui attributi sono le chiavi che contiene la mappa e i valori associati ad esse. La verifica consiste nell'effettuare un test in cui si calcola il tempo di esecuzione di entrambi i metodi, eseguiti sul tipo di mappa che usa la funzione `deserializeObject(...)`, all'interno della quale i metodi sono implementati. Il test fallisce se il metodo suggerito da Infer è più veloce di quello usato nel codice.

```

JsonObjectBuilder builder;
JsonObject obj;
Map<String, JsonValue> map = new HashMap<>();

@BeforeAll
void setUp()
{
    for(int i=0; i<10000; i++)
    {
        builder = Json.createObjectBuilder().add(Integer.toString(i), Integer.toString(i));
    }

    obj=builder.build();

    for(int i=0; i<10000; i++)
    {
        map.put(Integer.toString(i), Json.createValue(Integer.toString(i)));
    }
}

@Test
void testKeySet_1()
{
    long start1 = System.nanoTime();
    for (String name : obj.keySet())
    {
        map.put(name, obj.get(name));
    }

    long end1 = System.nanoTime();
    long duration1 = (end1 - start1);

    long start2 = System.nanoTime();
    for (Map.Entry<String, JsonValue> entry : map.entrySet())
    {
        map.put(entry.getKey(), entry.getValue());
    }

    long end2 = System.nanoTime();
    long duration2 = (end2 - start2);

    assertEquals(true, duration1 < duration2);
}

```

**Inefficient Keyset Iterator #2:** Valgono le stesse considerazioni fatte per la vulnerabilità #1. Di seguito il test effettuato.

```

@Test
void testKeySet_2()
{
    long start1 = System.nanoTime();
    for (String name : map.keySet())
    {
        JsonValue j=map.get(name);
    }

    long end1 = System.nanoTime();
    long duration1 = (end1 - start1);

    long start2 = System.nanoTime();
    for (Map.Entry<String, JsonValue> entry : map.entrySet())
    {
        JsonValue j = entry.getValue();
    }

    long end2 = System.nanoTime();
    long duration2 = (end2 - start2);

    assertEquals(true, duration1 < duration2);
}

```

**Null Dereference #4:** Infer segnala che l'oggetto ritornato da `unescapeJava(LiteralStringValueExpr.value)` può essere nullo ed è dereferenziato alla linea 99. Il metodo `unescapeJava` invoca a sua volta un altro metodo, `translate(value)` che ritorna null se `value` è null; `unescapeJava(value)` è chiamato dal metodo `asChar()` e la variabile `value` utilizzata viene inizializzata dal costruttore. Di seguito il test effettuato, che fallisce se viene generata una `NullPointerException`.

```
@Test
void testForNullDereference_4()
{
    String value=null;
    CharLiteralExpr c=new CharLiteralExpr(value);

    try
    {
        c.asChar();
    }
    catch (NullPointerException e)
    {
        fail("Null Pointer Exception 4");
    }
}
```

**Null Dereference #5:** Infer segnala che l'oggetto `annotations` può essere nullo ed è dereferenziato alla linea 63 dal costruttore `PackageDeclaration(...)`. `annotations` è inizializzata dal metodo `cloneList(n.getAnnotations(), arg)`, che ritorna null se `n.getAnnotations()` ritorna null. Quest'ultimo ritorna null se l'attributo `annotations` all'interno di `n`, che è un'istanza di `PackageDeclaration`, ha valore null. La funzione a cui facciamo riferimento, all'interno della quale è presente il codice problematico, è `visit(final PackageDeclaration n, final Object arg)`. Di seguito il test effettuato, che fallisce se viene generata una `NullPointerException`.

```
@Test
void testForNullDereference_5()
{
    Integer t=3;
    PackageDeclaration p=new PackageDeclaration();
    p.setAnnotations(null);
    CloneVisitor v=new CloneVisitor();

    try
    {
        v.visit(p, t);
    }
    catch (NullPointerException e)
    {
        fail("Null Pointer Exception 4");
    }
}
```

**Resource Leak #37 e #38:** Entrambe le vulnerabilità fanno riferimento alla stessa risorsa. Infatti, in questo caso Infer segnala che la risorsa di tipo `java.util.jar.JarFile` è stata acquisita alla linea 53, utilizzata alla linea 56 e 58 e non più rilasciata. Di seguito il test effettuato, con cui si invoca il metodo in questione e a prescindere dal risultato dell'esecuzione verifica se la risorsa è stata rilasciata o meno. In particolare, il test fallisce se non viene generata una `IOException` quando si prova ad effettuare un'operazione sulla risorsa in questione dopo l'esecuzione del metodo con cui viene acquisita. Perché se non viene generata, significa che l'operazione è andata a buon fine: cioè, la risorsa risulta ancora aperta.

```

@Test
void testResourceLeak_58() throws IOException
{
    File f=new File("/home/andre/App-Vulnerabili/glide-6.0.0-beta.2.aar");
    JarFile jarFile = new JarFile(f);
    JarTypeSolver delegate;

    try
    {
        ZipEntry classesJarEntry = jarFile.getEntry("classes.jar");
        if (classesJarEntry == null) {
            throw new IllegalArgumentException(String.format("The given file (%s) is malformed: entry classes.jar was not found", f.getAbsolutePath()));
        }
        delegate = new JarTypeSolver(jarFile.getInputStream(classesJarEntry));
    }
    finally
    {
        assertThrows(IOException.class, () -> {jarFile.getEntry("classes.jar");});
    }
}

```

## RISULTATI

Di seguito i risultati dei test eseguiti:

```

[ERROR] Failures:
[ERROR] InferTest.testKeySet_2:85 expected: <true> but was: <false>
[INFO]
[ERROR] Tests run: 13, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] Reactor Summary for javaparser-parent 3.25.0:
[INFO]
[INFO] javaparser-parent ..... SUCCESS [ 0.002 s]
[INFO] javaparser-core ..... SUCCESS [ 9.488 s]
[INFO] javaparser-core-serialization ..... FAILURE [ 1.892 s]
[INFO] -----
[INFO] BUILD FAILURE

```

```

[ERROR] Failures:
[ERROR] InferTest.testForNullDereference_4:19 A reference was unexpectedly null.
[ERROR] InferTest.testForNullDereference_5:36 A reference was unexpectedly null.
[INFO]
[ERROR] Tests run: 1897, Failures: 2, Errors: 0, Skipped: 7
[INFO]
[INFO] -----
[INFO] Reactor Summary for javaparser-parent 3.25.0:
[INFO]
[INFO] javaparser-parent ..... SUCCESS [ 0.002 s]
[INFO] javaparser-core ..... SUCCESS [ 9.422 s]
[INFO] javaparser-core-testing ..... FAILURE [ 36.994 s]
[INFO] javaparser-core-testing-bdd ..... SKIPPED
[INFO] javaparser-core-generators ..... SKIPPED
[INFO] javaparser-core-metamodel-generator ..... SKIPPED
[INFO] javaparser-core-serialization ..... SKIPPED
[INFO] javaparser-symbol-solver-core ..... SKIPPED
[INFO] javaparser-symbol-solver-testing ..... SKIPPED
[INFO] -----
[INFO] BUILD FAILURE

```

```

[ERROR] Failures:
[ERROR] InferTest.testResourceLeak_58:35 Expected java.io.IOException to be thrown, but nothing was thrown.
[INFO]
[ERROR] Tests run: 1680, Failures: 1, Errors: 0, Skipped: 26
[INFO]
[WARNING] Corrupted channel by directly writing to native stream in forked JVM 1. See FAQ web page and the dump file /home/andre/App-Vulnerabili/javaparser-javaparser-parent-3.25.0/javaparser-symbol-solver-testing/target/surefire-reports/2023-02-28T13-13-03_212-jvmRun1.dumpstream
[INFO] -----
[INFO] Reactor Summary for javaparser-parent 3.25.0:
[INFO]
[INFO] javaparser-parent ..... SUCCESS [ 0.003 s]
[INFO] javaparser-core ..... SUCCESS [ 9.303 s]
[INFO] javaparser-symbol-solver-core ..... SUCCESS [ 1.308 s]
[INFO] javaparser-symbol-solver-testing ..... FAILURE [ 54.784 s]
[INFO] -----
[INFO] BUILD FAILURE
[INFO]

```

Come si può notare, tutte le vulnerabilità testate sono valide.

Eccetto una, la #1: in questo caso, l'iteratore KeySet() implementato da chi ha sviluppato Javaparser sono è più efficiente rispetto al metodo EntrySet(), suggerito da Infer.