nccgroup

## NCC Group Whitepaper

# Accessing Private Fields Outside of Classes in Java

April 5, 2017

Prepared by
Robert C. Seacord

Abstract

Java developers are frequently unaware that the use of nested classes in Java programs weakens the accessibility guarantees of the language and allows private fields to be accessed from outside the class, potentially violating developers' assumptions and affecting overall security. The Java compiler weakens the accessibility of private members of an outer class when a nested inner class is present to package-private access. An attacker can create another class in the same package, which can access package-private classes, methods, and fields in the same package. This is particularly troublesome because of the increasing popularity of lambda expressions in Java 8 programming. This whitepaper describes the Java language mechanisms used in these exploits, specifies the extent to which the compiler weakens the accessibility of private fields, and identifies possible attack vectors.

# Table of Contents

# 1 Introduction

Java developers are frequently unaware that the use of nested classes in Java programs can weaken the language's accessibility guarantees and allow private fields to be accessed from outside their classes. The Java Language Specification [JLS 2015] allows classes and interfaces to be nested within each other. Within the scope of a top-level declaration, any number of types can appear nested, for example, as member types or inner classes. A top-level type along with all its nested types can be colloquially referred to as *nest mates*. Nest mates have unrestricted access to each other. This includes access to private fields, methods, and constructors. The private access is complete (undifferentiated, flat) within the entire declaration of the containing package.

This weakening of Java language guarantees can violate developers' assumptions about the accessibility of private fields, methods, and constructors within packages containing nested types, affecting higher-level layers of security [Lai 2008]. As a result, it is critical to understand specifically when this occurs and the exact risk. Rule "OBJ08-J. Do not expose private members of an outer class from within a nested class" of *The CERT Oracle Secure Coding Standard for Java* [Long 2012] attempted to do this, but contains some factual errors and omissions. This whitepaper is an attempt to correct and extend this imperfect rule. The remainder of this paper examines the Java language mechanisms involved in the overall language vulnerabilities, describes how attackers can access private fields from outside of their classes, and provides some background as to how this language vulnerability might be exploited before summarizing.

# 2 Java Language Mechanisms

This section describes the Java language mechanisms that allow private fields to be accessed from outside their classes.

## 2.1 Access Level Modifiers

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. Every class member has an accessibility of `public`, `protected`, package-private (no modifier),[1] or `private`. Table 1 enumerates the accessibility of class members. Private members can only be accessed by methods in the same class. Package-private access allows access by any method in the same package, but not by methods in different packages. Protected members are accessible by subclasses in the same package or by subclasses in other packages. Public members are accessible everywhere. For members of public classes, an increase in accessibility from package-private to protected is significant. A protected member is part of the class's exported API and potentially exposes these members to malicious subclasses.

| Modifier | Class | Package | Subclass | World |
|---|---|---|---|---|
| **public** | Y | Y | Y | Y |
| **protected** | Y | Y | Y | N |
| **no modifier** | Y | Y | N | N |
| **private** | Y | N | N | N |

Table 1: Access levels

It is a recommended best practice [Bloch 2008, Long 2012] that developers make fields as inaccessible as possible by using the most restrictive access level available. In particular, fields should be private unless there is a good reason to do otherwise. Private fields allow classes to hide internal data and other implementation details from other classes. This concept is known as *information hiding* or *encapsulation* and is one of the fundamental tenets of software design [Parnas 1972]. Joshua Bloch [Bloch 2008] describes the rationale for minimizing accessibility in *Effective Java*, 2nd edition, Item 13, "Minimize the accessibility of classes and members".

While programmers use access level modifiers to implement common software design principles, it is less clear how they help to improve the program's overall security. For example, private fields cannot prevent an attacker from disassembling class files using the `javap` disassembler or decompiling these files into usable Java source code. Almost any Java decompiler including Procyon,[2] CFR,[3] JD,[4] Fernflower,[5] Krakatau,[6] or Candle[7] can accomplish this task. An attacker with access to the class and JAR (Java ARchive) files can easily reconstruct the fields and logic of these classes. The ability to decompile Java classes, however, does not mean that an attacker can access the data stored in private fields at runtime.

If an attacker can create a malicious class in a package, that class can access package-private classes, methods, and fields in that same package. In packages containing nested classes, the information in private fields can be accessed using the Java Reflection API [API 2014], as demonstrated in Section 3. Reading private fields might allow information such as private keys, passwords, or credit card information to be leaked. Writing to private fields could break class invariants with unpredictable results.

[1] A.K.A, *package access* or *default access*
[2] https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler
[3] http://www.benf.org/other/cfr/
[4] http://jd.benow.ca/
[5] https://github.com/fesh0r/fernflower
[6] https://github.com/Storyyeller/Krakatau
[7] https://github.com/bradsdavis/candle-decompiler

The Java Reflection API includes the `getDeclaredFields()` method that allows a caller to access public, protected, package-private, and private fields but not inherited fields. This mechanism allows attackers to enumerate the private fields in a class using the following code:

```
// Returns an array of Field objects reflecting
// all the fields declared by the class (including private)
final Field fields[] = FieldTest.class.getDeclaredFields();

// Enumerate fields
for (int i = 0; i < fields.length; ++i) {
  System.out.println("Field: " + fields[i]);
}
```

Limited information leakage occurs in that the names of the fields are visible. Attackers may focus their attention on fields with suspect names such as `SecretPrivateKey`. In the following code snippet, a developer is leaking runtime data by naming the fields according to the data they contain, possibly in a misguided attempt to avoid the use of magic numbers that lack meaningful names:

```
public class Leak {
  private static final int Bx41 = 0x41;
  private static final int Bx42 = 0x42;
  private static final int Bx43 = 0x43;
  private static final int Bx44 = 0x44;
  private static byte[] keyBytes = new byte[] {Bx41, Bx42, Bx43, Bx44};
  private static SecretKeySpec key;
  private static Cipher cipher;
}
```

Code running in the same package with an installed security manager [JSO 2016] can discern a number of details of this implementation using reflection.

The following code,

```
final Field fields[] = Leak.class.getDeclaredFields();

// Enumerate fields
for (int i = 0; i < fields.length; ++i) {
  System.out.println("Field: " + fields[i]);
}
```

will output the following information about `class Leak`:

```
Field: private static final int Leak.Bx41
Field: private static final int Leak.Bx42
Field: private static final int Leak.Bx43
Field: private static final int Leak.Bx44
Field: private static byte[] Leak.keyBytes
Field: private static javax.crypto.spec.SecretKeySpec Leak.key
Field: private static javax.crypto.Cipher Leak.cipher
```

Note that the actual bytes stored in `keyBytes` and their order can be discerned inspecting the Java `.class` using `javap` or a Java decompiler, when the `.class` or `.jar` file is available.

Assuming that the developer makes no such mistakes and that a security manager is installed, trying to access a private field through reflection will result in an `IllegalAccesException`. To avoid this exception, the attacker must change the accessibility of the private field by calling the `setAccessible()` method for the field. Doing so invokes the security manager's `checkPermission()` method for the `ReflectPermission("suppressAccessChecks")` permission, which throws a `SecurityException` if the request is denied. Enabling this permission in this situation is a vulnerability; a well-configured application must not permit untrusted code to suppress access checks.

## 2.2 Nested Classes

The Java programming language allows a class to be defined within another class. These *nested classes* can be static or non-static. There are four kinds of non-static nested classes: inner classes, local classes, anonymous classes, and lambda expressions.

### Inner Classes

Non-static nested classes are called *inner classes*. Inner classes have access to other members of their enclosing classes, including private members. Objects that are instances of an inner class exist within an instance of the outer class. Consider the following classes:

```
class OuterClass {
  ...
  class InnerClass {
      ...
  }
}
```

An instance of `InnerClass` can exist only within an instance of `OuterClass` and has direct access to the methods and fields of its enclosing instance.

### Local Classes

There are two special kinds of inner classes: *local classes* and *anonymous classes*. Local classes are classes that are defined in a block, typically within the body of a method. Inner classes declared within the body of a method, without naming the class, are known as anonymous classes.

A local class has access to the members of its enclosing class. When a local class accesses a local variable or parameter of the enclosing block, it *captures* that variable or parameter. (A captured variable is one that has been copied so it can be used in a nested class. It must be copied because it may outlive the current context.) In addition, a local class declared in a method has access to local variables that are declared final. Starting in Java SE 8, a local class declared in a method can also access the method's parameters and any local variables that are *effectively final* (that is, their values are not changed after initialization).

### Anonymous Classes

Anonymous classes exist to make code more concise by declaring and instantiating a class at the same time. They are similar to local classes except that they are unnamed. While local classes are class declarations, anonymous classes are expressions that are defined within other expressions. The following example from *The Java Tutorials* [Tutorials 2016] uses anonymous classes in the initialization statements of the local variable `spanishGreeting`:

```
HelloWorld spanishGreeting = new HelloWorld() {
  String name = "mundo";
  public void greet() {
    greetSomeone("mundo");
  }
  public void greetSomeone(String someone) {
    name = someone;
    System.out.println("Hola, " + name);
  }
};
```

Similar to local classes, anonymous classes can capture variables and have the same access to local variables of the enclosing scope. An anonymous class has access to the members of its enclosing class, but it can only access local variables in its enclosing scope that are final or effectively final.

### Lambda Expressions

The syntax of anonymous classes can be unwieldy in cases where the interface contains only one method. Lambda expressions are nested classes that allow functionality to be passed as an argument to another method (that is, to treat functionality as a method argument or code as data). Lambda expressions can also capture variables; they have the same access to local variables of the enclosing scope as local and anonymous classes. Lambda expressions are lexically scoped, meaning that they do not inherit any names from a supertype or introduce a new level of scoping. Declarations in a lambda expression are interpreted in the same manner as in the enclosing environment. Consequently, fields, methods, and local variables of the enclosing scope can be accessed directly. Similar to local and anonymous classes, a lambda expression can only access local variables and parameters of the enclosing block that are final or effectively final.

# 3 Accessibility of Private Fields

When the Java compiler compiles nested classes, it creates classes, methods, fields, and other constructs that do not have a corresponding construct in the source code. The Java documentation for `Member.isSynthetic()` states that it returns "true if and only if this member was introduced by the compiler." Synthetic constructs enable Java compilers to implement new Java language features without changes to the Java virtual machine (JVM). Synthetic constructs, and their corresponding class files, vary among Java compiler implementations.

The following definition for class `Coordinates` from Rule "OBJ08-J. Do not expose private members of an outer class from within a nested class" of *The CERT Oracle Secure Coding Standard for Java* [Long 2012] defines two private fields `x` and `y` and a private inner class `Point` containing a single private method `getPoint()`:

```
class Coordinates {
  private int x;
  private int y;
  private class Point {
    private void getPoint() {
      System.out.println("(" + x + "," + y + ")");
    }
  }
}
```

The Java compiler compiles a group of nested types into a corresponding group of class files. Each nested type is *flattened* during compilation into a package member with an encoded name called its binary name [JLS 2015]. The encoding is unambiguously reversible with the help of the `InnerClasses` and `EnclosingMethod` class file attributes, as defined in the JVM Specification [JVMS 2015]. At the JVM level, the package-private access protection is the closest approximation to private access protection that is allowed between package members. Because nest mates are compiled to package members, the compiler must provide access to private names (outside of a single class) by creating various wrapper methods. These wrapper methods are synthetic and package-private. Decompiling the `Coordinates` class file with the `javap` class file dissembler reveals two synthetic methods:

```
$ javap -c Coordinates
static int access$000(OBJ08J.Coordinates);
    flags: ACC_STATIC, ACC_SYNTHETIC
    Code:
       0: aload_0
       1: getfield      #2        // Field x:I
       4: ireturn
  static int access$100(OBJ08J.Coordinates);
    flags: ACC_STATIC, ACC_SYNTHETIC
    Code:
       0: aload_0
       1: getfield      #1        // Field y:I
       4: ireturn
```

The Java compiler generates two synthetic methods: `access$000` returns the value of field `x`, and `access$100` returns the value of field `y`. The Java compiler introduces synthetic methods on an as-needed basis. When only one of the enclosing class's private fields is accessed by the nested class, only one synthetic method (`access$000`) is created by the compiler. However, when both private fields are accessed, two corresponding synthetic methods are generated by the compiler (`access$000`, `access$100`).

The `Coordinates` outer class and `Point` inner class are compiled to package members, and the synthetic accessor methods are used by the `Point` inner class to read the values of the two private fields from within the `Coordinates` class.

The compiler enforces proper access controls at compile time. Attempts to access fields private to `Coordinates` from classes other than `Coordinates` and `Point` results in a compilation error. However, the compiler-generated class files result in a different set of access controls being enforced at runtime. There is consequently no risk of trusted classes unintentionally accessing the newly designated package-private members. (While not a security risk, this could otherwise break encapsulation.) However, any package-private member is vulnerable if attacker-supplied code can infiltrate the package. A package attack involves adding new classes to a package, replacing existing classes in a package, or both. Such attacks are possible if trusted code is hosted in a JVM alongside untrusted code.

Even in the presence of an installed and operational security manager, attacker-supplied code running in the same package can retrieve the values of the private fields using the `getDeclaredMethod()` method from the Java Reflection API [API 2014] to invoke the synthetic accessor methods:

```
Coordinates ic = new Coordinates();
Method m = Coordinates.class.getDeclaredMethod(
  "access$000", Coordinates.class
);
Integer x = (Integer) m.invoke(null, ic);
m = Coordinates.class.getDeclaredMethod("access$100", Coordinates.class);
Integer y = (Integer) m.invoke(null, ic);
```

Classes loaded by different loaders do not have package-private access to one another even if they have the same package name. Classes in the same package loaded by the same class loader must either share the same code signing certificate or not have a certificate at all. In the JVM, class loaders are responsible for defining packages. Consequently, packages should be sealed in the JAR file manifest [SCG 2015] to ensure that new classes are not added to them. Without sealing, attacker-supplied code could create a class and define it to be a member of another package. The attacker-supplied software would consequently gain access to package-protected members in the vulnerable, unsealed package.

A JAR is sealed by adding the `Sealed` header to the manifest of the JAR file containing the package. A sealed JAR specifies that all packages defined by that JAR are sealed unless overridden specifically for a package. Individual packages are sealed by associating a `Sealed` header with the package's `Name` header. A `Sealed` header not associated with an individual package in the archive signals that all packages are sealed. These `Sealed` headers are overridden by any `Sealed` headers associated with individual packages. The value associated with the `Sealed` header is either `true` or `false`. If a package is sealed, all classes defined in that package must originate from a single JAR file or a `SecurityException` is thrown.

Attackers can exploit deserialization to create an arbitrary object, provided that the object's class is available on the `classpath` specified for the JVM. This problem is described by Rule "SER12-J Prevent deserialization of untrusted data" and related rules in *The CERT Oracle Secure Coding Standard for Java* and the *Java Coding Guidelines* [Long 2013]. The closest CWE is "CWE-502: Deserialization of Untrusted Data." Java deserialization vulnerabilities are a serious problem, but not the topic of this whitepaper. The ability of attackers to access a class's private fields from attacker-created objects exacerbates this problem. However, eliminating the ability of an attacker to access a class's private fields does not eliminate the threat posed by Java deserialization vulnerabilities. Consequently, dealing with potential deserialization vulnerabilities should be a priority, but developers should also be on the lookout for situations in which the use of nested classes violates their security assumptions.

# 5 Conclusion

The use of nested classes in Java programs weakens the accessibility guarantees of the language and allows private fields to be accessed from outside the class, potentially violating developers' assumptions and affecting overall security. The Java compiler weakens the accessibility of private members of an outer class when a nested inner class is present to package-private access. An attacker can create another class in the same package, which can access package-private classes, methods, and fields in the same package. This is particularly troublesome because of the increasing popularity of lambda expressions in Java 8 programming.

Minimally, developers should understand that inner classes can weaken the accessibility of private fields. If the security of an application depends on these fields remaining private, it may be necessary to eliminate the use of inner classes or otherwise redesign the application so the risk of accessing these fields from untrusted code is eliminated. It would be beneficial if the JVM checks were better aligned with the Java language rules for nested classes.[8]

[8] http://openjdk.java.net/jeps/181

# 6 Acknowledgements

Thanks to Jeremy Brandt-Young, David Goldsmith, Andy Grant, Heather Overcash, Audrey Saunders, for supporting this effort. Thanks to my technical reviewers Rennie deGraaf, Jake Heath, Fred Long (Aberystwyth University), Steve Park, Milton Smith (Oracle), and Justin Taft.

# 7 Author Bio

Robert C. Seacord, a renowned computer scientist and author, known as the "father of secure coding." Robert is a Principal Security Consultant with NCC Group where he works with software developers and software development organizations to eliminate vulnerabilities resulting from coding errors before they are deployed. Previously, Robert led the secure coding initiative in the CERT Division of Carnegie Mellon University's Software Engineering Institute (SEI). Robert is also an adjunct professor in the School of Computer Science and the Information Networking Institute at Carnegie Mellon University. Robert is the author of six books, including *The CERT C Coding Standard, Second Edition* (Addison-Wesley, 2014), *Secure Coding in C and C++, Second Edition* (Addison-Wesley, 2013), *The CERT Oracle Secure Coding Standard for Java* (Addison-Wesley, 2012), and *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* (Addison-Wesley, 2014). Robert is on the Advisory Board for the Linux Foundation and an expert on the ISO/IEC JTC1/SC22/WG14 international standardization working group for the C programming language.

# 8 References


[API 2014] Java Platform, Standard Edition 8 API Specification, Oracle (2014).

[Bloch 2008] Bloch, Joshua. *Effective Java*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley (2008).

[JSO 2016] Java Security Overview, Oracle (2016).

[JLS 2015] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. Java Language Specification: Java SE 8 Edition. Oracle America (2016).

[JVMS 2015] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley. The Java Virtual Machine Specification: Java SE 8 Edition. Oracle America (2015).

[Lai 2008] C. Lai, "Java Insecurity: Accounting for Subtleties That Can Compromise Code," in *IEEE Software*, vol. 25, no. 1, pp. 13-19, Jan.-Feb. 2008. doi: 10.1109/MS.2008.9

[Long 2012] Long, Fred, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*, SEI Series in Software Engineering. Boston: Addison-Wesley (2012).

[Long 2013] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. 2013. *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* (1st ed.). Addison-Wesley Professional.

[Parnas 1972] Parnas, D. L. On the Criteria to Be Used in Decomposing Systems into Modules. In *Communications of the ACM* 15 (1972): 1053–1058.

[SCG 2015] Secure Coding Guidelines for Java SE, version 5.1 Oracle (2015).

[Tutorials 2016] The Java Tutorials. Oracle (2016).