Privacy Leaks in Java Classes

Jacob Sparre Andersen

Jacob Sparre Andersen Research & Innovation

June 2014

Privacy Leaks in Java Classes

DO-178C makes it important to study the reliability deficiencies of Java – and learn how to avoid them.

This presentation is focused on one particular class of programming errors in Java; **privacy leaks** from encapsulated data structures.

In Java assignment (=) works differently for simple and composite types (classes). One of the consequences of this design decision in Java is that “getters” for composite type attributes have to be written differently than those for simple type attributes.

Definition: Privacy leaks

Privacy leaks in general:

When somebody “outside” gets a copy of an object meant to be securely “inside”... [1]

In our specific context:

When a client of a Java class can **modify** the (referenced) value of a **private attribute** of the class.

[1] http://www.eecs.yorku.ca/course_archive/2012-13/F/1030/announcements/1030-07.pdf

Potential bugs due to privacy leaks

When a class is implemented with private attributes, it can be to:

- Enforce a consistent state inside an object of that class.
- Make objects of that class immutable.

A privacy leak will break these possible intended features of making attributes private.

An “interesting” side-effects could for example be that a client accidentally modifies encryption parameters, making the resulting encryption trivial to break.

Example of safe “getter”

```
public class NonLeaking {
   private Counter leakedCounter;
```

...

```
   /* Getter written by hand: */
   public Counter getLeakedCounter() {
      return new Counter(leakedCounter);
   }
```

The **new** operator creates a new object. Here of class `Counter` and as a copy of the private attribute `leakedCounter`.

Full source code: `http://repositories.jacob-sparre.dk/privacy-leaks-examples/`

Example: Generating a “getter” in Eclipse

Lets see how the state-of-the art Java IDE, Eclipse, thinks a “getter” for this class should be written...

Example of privacy leaking “getter”

```
public class Leaking {
   private Counter leakedCounter;
```

...

```
   /* Getter generated by Eclipse: */
   public Counter getLeakedCounter() {
      return leakedCounter;
   }
```

Notice how the “getter” generated by Eclipse simply returns a copy of the **reference** to the private attribute; giving the client access to modify the internal state of an object in the class directly.

Full source code: `http://repositories.jacob-sparre.dk/privacy-leaks-examples/`

The pattern of a privacy leaking “getter” (step 1)

Step one is to identify by-reference, private attributes. Only these are subject to the kind of privacy leaks we are looking for.

In the normalised sources files we work on, this reduces to identifying lines of the form:

“`private`” *type_identifier* *attribute_identifier* “`;`”

We are interested in the matching attributes, whose type isn’t a simple type or an immutable class.

The pattern of a privacy leaking “getter” (step 2)

Step two is to identify cases where a method returns a simple copy of an attribute. These are the source of the kind of privacy leaks we are looking for.

In the normalised sources files we work on, this reduces to identifying lines of the form:

“`return this.`”*attribute_identifier*“`;`”

where the attribute is on our list from step one.

The pattern of a privacy leaking “getter” (step 3)

In case step two finds matching lines (i.e. potentially leaking “getters”), step three is to see if the class of the potentially leaked attribute is actually immutable, as this makes a simple “getter” safe.

If the class isn’t already known to be mutable, a manual inspection of the class is required at this point.

Any “getters” which match after this check must be considered privacy leaking, and thus unsafe.

A real-life example

(a real-life example from a security-critical application)

Comparison with Ada

The most relevant difference between Ada and Java with respect to the kind of privacy leaks I have presented here is:

- Assignment in Java works differently for simple and composite objects (“classes”), whereas Ada has consistent assignments (but requires the programmer worry about object or reference to object).

This means that a **superficially equivalent** record type and “getter” written in Ada will be **safe** with regard to privacy leaking.

Comparison with Ada (continued)

Ada does have to option of storing references to attributes – as Java does it behind the scenes – but has to be done explicitly by the programmer.

It is my impression that the explicitness required by Ada is enough to make programmers aware of how the attribute should be handled, but it is not something I have hard documentation of.

Contact

Jacob Sparre Andersen
Jacob Sparre Andersen Research & Innovation
jacob@jacob-sparre.dk
http://www.jacob-sparre.dk/

You can find my Open Source software repositories at:
http://repositories.jacob-sparre.dk/