EXECUTION PROCESS IN JAVA

Jyoti yadav
Student, Department of Computer Engineering & Information Technology, Veermata Jijabai Technological Institute, Mumbai, Maharashtra, India

ABSTRACT

In this paper first of all we will describe that how we will execute our program in Java. The most typical use of reflection by far is for creating new objects given the object class name. The most common usage idiom for reflectively creating an object. Reflective APIs in Java are used for object creation, Method invocation, and field access, as described below.

INTRODUCTION

OBJECT CREATION:
Object creation APIs in Java provide a way to programmatically create objects of a class, whose name is provided at runtime; parameters of the object constructor can be passed in as necessary. Obtaining a class given its name is most typically done using a call to one of the static functions Class.forName(String, ...) and passing the class name as the first parameter. We should point out that while Class.forName is the most common way to obtain a class given its name, it may not be the only method for doing so. An application may define a native method that implements the same functionality. The same observation applies to other standard reflective API methods. The commonly used Java idiom T.class, where T is a class is translated by the compiler to a call to Class.forName(T.getName()). Since our reflection resolution algorithm works at the byte code level, T.class constructs do not require a special treatment. Creating an object with an empty constructor is achieved through a call to newInstance on the appropriate java.lang.Class object, which provides a runtime representation of a class.

Method Invocation:
Methods are obtained from a Class object by supplying the method signature or by iterating through the array of Methods returned by one of Class functions. Methods are subsequently invoked by calling Method.invoke.

Accessing Fields:
Fields of Java runtime objects can be read and written at runtime. Calls to Field.get and Field.set can be used to get and set fields containing objects. Additional methods are provided for fields of primitive types.

3 Assumptions About Reflection
This section presents assumptions we make in our static analysis for resolving reflection in Java programs. We believe that these assumptions are quite reasonable and hold for many real-life Java applications. The problem of precisely determining the classes that an application may access is undecidable. Furthermore, for applications that access the network,
1. String className = ...;
2. Class c = Class.forName(className);
3. Object o = c.newInstance();
4. T t = (T) o;
Fig. 2: Typical use of reflection to create new objects.
the set of classes that may be accessed is unbounded: we cannot possibly hope to analyze all classes that the application may conceivably download from the net and load at runtime. Programs can also dynamically generate classes to be subsequently loaded. Our analysis assumes a closed world, as defined below. Assumption 1. Closed world. We assume that only classes reachable from the class path at analysis time can be used by the application at runtime. In the presence of user-defined class loaders, it is impossible to statically determine the behavior of function Class.forName. If custom class loaders are used, the behavior of Class.forName can change; it is even possible for a malicious class loader to return completely unrelated classes in response to a Class.forName call. The following assumption allows us to interpret calls to Class.forName. Assumption 2. Well-behaved class loaders. The name of the class returned by a call to Class.forName(className) equals className. To check the validity of Assumption 2, we have instrumented large applications to observe the behavior of Class.forName; we have never encountered a violation of this assumption. Finally, we introduce the following assumption that allows us to leverage type cast information contained in the program to constrain the targets of reflective calls. Assumption 3. Correct casts. Type cast operations that always operate on the result of a call to newInstance are correct; they will always succeed without throwing a ClassCastException. We believe this to be a valid
practical assumption: while it is possible to have casts that fail, causing an exception that is caught so that the instantiated object can be used afterwards, we have not seen such cases in practice. Typical catch blocks around such casts lead to the program terminating with an error message.

References:
1. From the java book.
3 Through the internet.