Mechanisms for Secure Modular Programming in Java

Lujo Bauer Andrew W. Appel Edward W. Felten
presented by Jun He

1 Introduction

Java packages have limited ability to control access to their member classes, they don’t have explicit interfaces, and they don’t support multiple views of modules. These make the Java package system deficient for modular programming tasks in which security is important; for example, for writing mobile applications.

This paper presents a new module system for Java that improves upon the deficiencies of the Java package system and gives the programmer more control over dynamic linking. The module system provides explicit interfaces, multiple views of modules based on hierarchical nesting, and more flexible namespace management than the Java package system. Relationships between modules are explicitly specified in module description files. More control can be provided over dynamic linking by allowing import statements in module description files to require that imported modules be annotated with certain properties, which are implemented by digital signatures. The module system is compatible enough with standard Java that it has been implemented as a source-to-source and bytecode-to-bytecode transformation wrapped around a standard Java compiler.

2 hierarchical module system

The module system, like Java packages, groups classes into larger units. A module in this system consists of a set of source files and a module description file. The module description file consists of three parts:

- an export interface;
- a membership list;
• a set of import statements.

The export interface of the module is a filter that allows only select classes to be visible externally. The source files that comprise the module are listed in the membership list. The only way to reference classes that are not in the module is through the import interface.

3 Fixing Java Packages

The module system contains a number of features that either are not present or are insufficiently developed in the Java package system. The most important are explicit export interfaces and membership lists, hierarchical scalability and multiple interfaces, and convenient namespace management. In addition to their value as software engineering tools, these are all instrumental in forming a base for developing secure software systems in Java.

Export Interface and Membership Lists The Java package system is unsuited for modular programming and enforcing security policies. The combination of implicit interfaces and the lack of explicit membership lists would make it easy for a malicious attacker to take advantage of a system for running mobile code that based its security facilities on Java packages. The module system prevents any such security breach by using module description files which explicitly specify both the membership of a module and its public interface by listing all the classes that belong to each.

Hierarchical Scalability and Multiple Interface The module system has the ability to structure modules so as to provide different views to different clients, which can't be provided by Java package system. The module system supports hierarchical modularity by allowing modules to explicitly list the submodules on which they depend. Modules can export not only classes that have been defined in their own source files, but also classes that have been defined in imported modules.

Name-Space Management An additional software engineering benefit is the module system's flexible and convenient namespace management scheme. Java package system try to avoid name clashes between classes by grouping code into packages. But Java packages are themselves named, so that merely lifts the problem to the package level. The modules in the module system, on the other hand, are not named, so they don't suffer from this problem. Modules are assigned names only via import statements of individual module description files; this type of namespace thinning makes it easy to keep their names short and simple.
4 Secure Linking

The behavior of a program fragment depends not only on its own code but also on the libraries with which it is linked. Therefore, safety guarantees are needed when a system must trust the behavior of a particular executable, such as an applet. Java often uses code signing for such purposes. But both the theory and the practice of code signing are insufficiently developed. The module system makes headway on this by allowing the programmer to require certain properties of the modules on which his code depends. If the required properties are not present, the program won’t link or execute. If they are present, the programmer can more realistically expect that his program will behave in the desired manner.

5 Implementation

A prototype implementation has been developed to illustrate the features of the module system. Modules can be translated into Java packages. Some of the features of the module system, however—in particular its ability to place constraints on linking—cannot be expressed just using Java bytecode. Therefore, the prototype needs to provide additional features both to the compiler and to the virtual machine.

Compilation The compilation phase is a wrapper around a standard Java compiler that consists of a preprocessing and a postprocessing step. The job of the preprocessing is to translate the source code into equivalent standard Java source code. The compilation phase also has a postcompilation step. Modules can export symbols that have been defined in imported modules, so it is possible that several module description files need to be traversed to discover to which class a qualified identifier is pointing.

Execution in the Virtual Machine Each module description file sets up a mapping from identifiers to the classes they represent. The same identifier can therefore represent different classes in different modules. A request to load a class, too, may be allowed or denied depending on whether the class is signed by the digital signature required by the calling module. Therefore, the JVM should have the ability to answer loadClass requests differently depending on the module from which they originate. The solution is to extend the ClassLoader class. A new copy of this extended class loader is instantiated for every module that is loaded by the VM. The class loader uses the module description file to set up the appropriate class environment and control linking in the manner specified by export filters and digital signature
requirements.

**Name Hacking** The full name of every compiled class is encoded in its bytecode. The VM verifies that the encoded name of a requested class matches the name with which loadClass was invoked. Class names in the module system contain identifiers defined in module description files; these names may bear little relation to the actual package names assigned to the classes. With the security checks, it is not possible for the class loader to naively redirect loadClass requests to classes whose names don't match the requested ones. The solution is to rewrite the bytecode of compiled modules, replacing symbolic names (defined through module description files) with actual ones. This is done while a class is being loaded into the VM, before linking or bytecode verification.

**The Reflection API** The modular system interacts badly with Java's reflection API. The security features of class loaders require that the implementation of the naming scheme differ considerably from the view presented to the programmer. This renders the forName and getName methods useless. But it is it is dubious whether the Java's reflection API should be available for public use.

## 6 Conclusions and Future work

The module system is based on explicit module descriptions. Membership lists and explicit export interfaces protect module integrity. Unnamed modules and declarative import statements provide simple and convenient namespace management. Variable levels of access to modules are supported by arranging modules in hierarchies. Increased control over the linking process, implemented by allowing import statements to specify digital keys for imported modules, helps guarantee correct program behavior in the presence of dynamic linking.

Any attempt to develop a secure programming environment is likely to be based on a module system. The reflection API is a serious obstacle to sophisticated module systems that support nesting and reexporting and have opaque interfaces. Dynamic linking is an area that deserves more study. It is important to provide guarantees about the behavior of dynamically linked libraries. The module system provides a good framework for annotating code with such guarantees.