Abstract

This paper describes an extension of the Java language that provides “programmable security.” The approach augments the Java syntax with constructs for specifying various access control policies for Java packages, including DAC, MAC, RBAC and TBAC. A primitive ticket based mechanism serves as the foundation for programmable security. The implementation incorporates a preprocessor for language translation and a security service library that implements the ticket management infrastructure. The preprocessor translates the extended Java source code to native Java for eventual bytecode interpretation, simultaneously binding security services to the native code. The design is simple and flexible, and provides developers with an effective tool for “programming” security within Java packages.

1. Introduction

Programmable security is a promising approach for achieving high assurance in open, distributed environments. The idea is to embed primitive security mechanisms, and constructs for programming and verifying security within distributed object languages and architectures. This approach is similar to the incorporation of primitive data types, type constructors and type-checking facilities in conventional programming languages. Just as strong typing improves code reliability, programming and checking for security at the language level can significantly enhance the security and reliability of software systems deployed in open environments.

The platform-independence and object-oriented features of Java have made it the language of choice for Internet applications. However, security was not a prime consideration in the original design. Indeed, the evolution of Java has been largely motivated by the need for enhanced security features.

This paper enhances Java by augmenting its “all or none” package protection scheme with a special syntax that permits programmable access control, including DAC, MAC, RBAC and TBAC. Developers can use the extended syntax to specify fine-grained access control for individual Java packages. Note that this approach is designed to complement – not replace – the Java security model.

A primitive ticket based mechanism serves as the foundation for programmable security. The implementation uses a preprocessor for language translation and a security service library that implements the ticket management infrastructure. The preprocessor translates the extended Java source code to native Java for eventual bytecode interpretation, simultaneously binding security services to the native code. In addition, this approach supports the seamless integration of legacy Java code. The design is simple and flexible, and provides developers with an effective tool for “programming” security within Java packages.

The following section describes the package protection strategy, the authorization model, extensions to the Java grammar, and the execution model. Next, the design and implementation are presented. Message handlers, package classes, method and field access, interfaces and code compatibility are discussed in detail.

2. Java package access control extensions

This section describes the basic principles and mechanisms for implementing package based access control in Java (JPAC). In particular, it discusses the relationship between the Java security model and the JPAC protection model. The formal authorization model used to implement fine-grained access control for Java packages is also presented.

2.1. Java package protection model

Packages provide Java developers with a programmatic tool for grouping classes and interfaces. Packages embody namespace separation for code and effectively limit access between programming elements across organizational boundaries [2]. Java protection settings, applied to classes,
interfaces, methods and fields, establish constraints on accessibility based in part on package organization. Java package structures and protection settings define access control in Java programs.

Java elements can be assigned one of four protection levels: public, protected, default and private. Public elements are visible to all active Java program classes and elements. Protected elements are visible within their packages and within subclasses. Default elements are visible within their packages and private elements are visible only within their own classes.

Non-private class members within the same Java package are assumed to share a level of trust, and are thus visible to each other. Access across package boundaries is possible only for public (or protected) class elements.

Several features complicate the Java protection model. Following object-oriented conventions, Java subclasses inherit superclass elements that are labeled as protected. But these can only be accessed through subclass instances, and not directly at the superclass level. Moreover, Java classes not explicitly placed in a package are grouped in a default “unnamed” package.

Unfortunately, the Java security model has some limitations, most notably, reduced flexibility and expressiveness. This makes it difficult to specify and enforce sophisticated security policies for many enterprises.

Figure 1 illustrates such a situation. The object hierarchy for a medical enterprise comprises four elements: (i) a root object (Hospital), (ii) administrative personnel (Admin), (iii) doctors (Doctor) and (iv) patients (Patient). Each Patient is described by clinical information (CInfo) and personal information (PInfo). The security policy allows a Doctor to access all Patient information but limits an Admin only to personal information (e.g., name, gender and billing address).

Unfortunately, standard Java provides all-or-nothing access across package boundaries. Therefore, if Patient information is to be accessed by classes in other packages (e.g., Admin and Doctor), classes in the package Patient would have to declare public. But this would make the information available to other classes, regardless of package. On the other hand, declaring Patient classes and fields as protected would only allow access to classes in the Patient package, not to classes in the Doctor package.

Our approach extends Java by providing lock and key primitives for fine-tuning security policies. Specifically, we use named locks within Patient to implement the hospital policy. Thus, a Doctor class may change the medication for a Patient class by sending a message dosageChange to an instance of CInfo using the access key doc. As shown in Figure 1, Hospital will accept the message since doctors have access to patients and an appropriate key is presented. When the message reaches Patient, the access permission is checked again before the message arrives at its intended destination (CInfo). An Admin instance cannot access CInfo because it does not possess the appropriate key.

2.2. Authorization model

The Java Package Access Control (JPAC) scheme described in this paper relies on a simple ticket based authorization model that is derived from more general models [6,12,13]. The simplified model considers only direct access permissions, not higher-order privileges (e.g., grant-grant and grant-revoke). Nevertheless, it is expressive and flexible, and is well suited to message passing systems, and Java in particular.

As described in Figure 1, messages requesting access to remote resources are passed between objects in a hierarchy. Note that any “object” in a hierarchy can also play the role of a subject. Tickets embedded in messages are analogous to capabilities [7,8,15,16] that convey privileges of message originators.

The authorization model is summarized in Figure 2. Rule 1 formally defines object adjacency relationships in a hierarchy. Since message passing occurs only between direct ancestors/descendants, message delegation implements “long-distance” communication in the hierarchy. Every message must be authorized for delegation at every intervening object in the hierarchy. Thus, messages are always passed between adjacent objects (predicate Adj). Adj s o1 o2 holds whenever objects o1 and o2 are adjacent in the hierarchy at state s. Parent s o1 o2 is true when o1 is the parent of o2 in state s.

Rule 2 defines the matching of “keys” and “locks” required to authorize access requests. Conceptually, tickets represent keys held by subjects that match locks placed on objects. The predicate Key s o1 t is true when o1 has a key named t in state s. Lock s o1 o2 t is true when o1 has a lock named t on object o2 in state s. Note that the hierarchy mandates that o1 and o2 be adjacent (Adj) for o1 to hold such a lock. The predicate Match defines when a message has permission to access o3 on behalf of o1 in o2.

Rules 3 and 4 comprise an inductive definition for access between two objects in a hierarchy. Rule 3 indicates that objects can access themselves; it represents the base case for inductive access checking. The predicate Access s o1 o2 specifies that o1 can access o2 from its point of origin in state s. Rule 4 provides the inductive step, stating that if o1 can access o2, and if o1 holds a key matching a lock in o2 for o3, then o1 can access o3.
2.3. Extended syntax

The Java Package Access Control (JPAC) implementation adds keywords to the Java grammar that specify access constraints for packages. The keyword unsecured is used in lieu of a package name to denote that a protected package is open to all packages. The default mode, explicitly named with the guarded keyword, makes a protected package visible to JPAC elements only.

Figure 3 presents the JPAC syntax extensions that express package based protection. These extensions build on the syntax described in the Java language specification documents [2,20,21,22].

The specification in Figure 3 changes how a compilation unit is named by making PackageDeclaration mandatory and adding a Properties production to it. Unnamed compilation units are specified by declaring a nameless package.

Three examples of legal compilation units are shown at the bottom of Figure 3. A package doctor specifies that its elements can be accessed by a package named admin. Associating the keyword guarded with the patient package specifies that its elements can be accessed by any other package using the JPAC protection system. The package other (labeled unsecured) specifies that any package, including native Java code, can access its elements. Unsecured packages are primarily used to interface with APIs and existing Java code.

The JPAC protection scheme considers access to include method calls or replies, and reads or writes on fields of a class or object. Other types of access, such as synchronization clauses and exceptions, are not controlled in this design. Public, private, protected and package-level protection modes are enforced as usual, with the package based protection specifications imparting additional authorization constraints.

2.4. Execution model

The JPAC semantics are derived from the authorization model described in Section 2.2. Every JPAC system defines an object hierarchy, consisting of a root object, and within it a collection of objects mapped to JPAC-protected packages. Classes and instances are regarded as components of their enclosing packages use package objects in the hierarchy to and pass messages conveying access requests/replies across protection boundaries. Figure 4 illustrates the organization of JPAC program elements.

Each protected package is associated with a unique token. A key-lock pair is generated from the token for each of the
packages listed in the protection declaration clause. Keys and locks are maintained in access control lists held by their parent packages. A special token conveying membership in the JPAC environment is used to define a key for distribution to all JPAC objects. Packages with guarded protection status hold the matching lock, allowing access by all JPAC objects, but not by external Java code.

The access mechanism is illustrated in Figure 4. A message originating in class immunologist is to be sent to an object in package patient. Class immunologist first constructs the message and sends it to its package message handler where it is checked for authorization. If it is authorized, the message is sent to the hospital message handler. The root class hospital then authorizes the message for delegation and delivery to patient. The patient package can grant or deny access. If access is granted, patient forwards the message to the destination program element for further processing.

The JPAC execution model supports message passing in a hierarchy of handlers to validate inter-package access. Keys and locks provide a mechanical basis for constraining access, while messages encapsulate authorization requests. A simple protocol helps guarantee their authenticity by ensuring that a lock and its matching key are created simultaneously. Locks in JPAC are never transported. However, keys may be given to other principals to implement transitive trust.

The centerpiece of the JPAC execution model is the message handler. Object hierarchies are modeled by establishing fixed communication pathways between message handlers. Messages are used only for authorization and consist of a key ring, source and destination identifiers, and a certificate. Key rings that accompany messages hold keys that embody the rights of the message originator. Messages are passed between objects via resident message handlers, authorized at each stop, until they reach their final destination.

3. JPAC implementation

Figure 5 illustrates JPAC’s extensional approach to implementing programmable security. This technique, inspired by [3], introduces two elements to a programming system: a preprocessor and a security service library. The preprocessor employs a parser for the augmented Java grammar to accept expressions in the extended language.
The augmented grammar adds production rules and keywords to the original grammar, binding security services to the programming system. The preprocessor translates the extended source code into native Java source code for eventual bytecode interpretation, simultaneously binding security services to the native code.

The preprocessor uses the security service library API to produce native Java source code that implements the desired security functionality. The resulting Java source is then passed to a native compiler that produces standard bytecode. The security service library can be linked at compile-time or run-time and perform its own security checks during execution.

The JPAC security service library class hierarchy is shown in Figure 6. The run-time ticket management is implemented with the classes shown inside the JPACEngine class library. In general, these classes handle run-time access control for JPAC code. To control access to unprotected methods in the class Object, all JPAC-protected classes inherit from the class JPACObject. Classes not protected by the JPAC system inherit data and code directly from Object.

3.1. Message handlers and package classes

The JPAC implementation instantiates a MessageHandler class and a class for each package to dispatch access control in object hierarchies. Instances of MessageHandler are associated with each newly-created “package class.” Package classes hold package-specific data such as the package name used to communicate between the message handlers, a reference to their MessageHandler instance, and JPAC protection settings. Package classes also initialize a fresh MessageHandler instance whenever a new JPAC protected class is loaded in the system.

Static methods in MessageHandler model behavior for the root JPAC domain, which serves as a secure intermediary between packages. A static code block creates a new message handler for each package class. The MessageHandler constructor contains code to obtain a reference to program units above it in the hierarchy. If the message handler obtains a reference to a unit that has not yet been instantiated, that unit’s static code will execute, creating its parent object. This produces a domino effect when a package is first accessed: message handlers for the package and every other package above it in the hierarchy are initialized.

Prior to each access, the checkOut() method of the message handler is invoked. This method traverses the object hierarchy to construct a message certificate. Keys and locks are checked at each node in the message path. If permission to proceed is granted, the local message handler digitally signs the certificate stored in the message and passes the message along. When a message successfully reaches its destination, it is returned through the chain of message handlers to its source. The message is then passed as a parameter to the callee method. The checkIn() method is executed within the callee method to traverse the object hierarchy backwards and verify the certificate’s integrity.

Figure 7 shows extended Java source code for a JPAC-protected AZTdata class. The class AZTdata is placed in the package patient and is intended to be used in studies involving the use of AZT in HIV-infected patients. Access to this class is restricted to classes in the doctor package to prevent unauthorized access (specified by the first clause in Figure 7). Class AZTdata is implemented by extending the class medication residing in package tools (this package contains various classes for processing medical information).

This simple example illustrates the efficacy of the access control methodology. Class AZTdata contains: (i) method addPatient that is invoked to add new patients to the AZT research study, (ii) method getNoOfPatients and (iii) instance variables noOfPatients and projectCode to count patients in the study and projectCode to identify the study. Note that the class AZTdata inherits noOfPatients
Figure 6. JPAC security service library.

Figure 7. Package based Java protection code (augmented syntax).

and getNoOfPatients from tools.medication and access to projectCode is set to be private.

Figure 8 shows the corresponding Java (package class) source code produced by the preprocessor from the patient package. Package classes are named as $N_{package\_name} \rightarrow “d” + flatten(N_{package\_name})$ to avoid clashes with existing code. These flattened names are produced by replacing all dots with the string “\textit{d}” and all underscores with a string of two underscores. JPAC classes and interfaces have their names similarly modified as $N_{class\_name} \rightarrow “d” + flatten(N_{class\_name})$.

Package class initialization involves the acquisition of a fresh MessageHandler. New message handlers are requested by calling the $\text{getMh()}$ static method in the MessageHandler class. Arguments to $\text{getMh()}$ indicate the name of the current package and the names of the packages to which access is allowed.

Only trusted software components – locally generated package classes and package classes generated by trusted entities – can invoke $\text{getMh()}$. A Java permission called JPACFreshMessageHandler constrains access to $\text{getMh()}$. Sample initialization code for the AZTdata class is shown in Figure 9.

3.2. Method and field access

The JPAC preprocessor transforms JPAC source code into native Java code for the purposes of authenticating calling subjects, and effecting secure method dispatch and field access. Method and field access are validated with a certificate placed as an extra parameter generated by the preprocessor (illustrated in the $\text{addMsg()}$ method in Figure 10). The preprocessor also integrates Java interfaces and unsecured packages with JPAC systems.

JPAC method calls are transformed to include an authenticating certificate as an extra argument (Figure 10). Key and Lock classes inherit from the Token class to model ticket based authorization with a $\text{match()}$ method. The $\text{match()}$ method checks keys and locks by computing an encrypted key value for comparison analogous lock values. MessageHandler::checkOut() performs authorization checks and returns certificates to the callee methods. Callee methods validate certificates with the MessageHandler::checkIn() method.

Unfortunately, the name of the destination cannot always be determined statically. Thus, two methods, $\text{getJPACNameDynamic()}$ and $\text{getJPACNameStatic()}$, are generated for every protected class. These methods help retrieve the name of the destination package for static access to parts of a class and for dynamic accesses from references.

Extending the Java protection model to provide package based access control provides a unique set of challenges to the preprocessor. For example, protected classes may invoke unsafe methods in inherited classes. Protected classes in JPAC inherit from the JPACObject class, which provides safe replacements for methods in
Patient package class.

```java
package patient;
import edu.utulsa.mcs.jpacengine.MessageHandler;
public final class _ppatient {
    private _ppatient() {} // Constructor.
    static MessageHandler mh;
    static String[] getProtection() {
        String[] result = {"hospital.patient", "hospital.doctor"};
        return result;
    }
    static MessageHandler getMh() {
        return mh;
    }
    static {
        mh = MessageHandler.getMh("hospital.patient", getProtection());
    }
    static public String getJPACName() {
        return "hospital.patient";
    }
}
```

Constructor-inheritance mapping.

```java
package patient;
public class _cAZTdata extends tools._cmedication {
    // the public method getNoOfPatients() and the field
    // public int noOfPatients are inherited from class tools.medication
    public int _anoOfPatients_ptools_dmedication(edu.utulsa.mcs.jpacengine.Certificate _temp0) {
        _ppatient.getMh().checkIn(_temp0);
        return super._anoOfPatients_ptools_dmedication(_ppatient.getMh().checkOut(super.getJPACNameDynamic()));
    }
    public int _anoOfPatients_ptools_dmedication(int _temp1, _ppatient.getMh().checkOut(super.getJPACNameDynamic()));
    }
    public int _mgetNoOfPatients(edu.utulsa.mcs.jpacengine.Certificate _temp0) {
        _ppatient.getMh().checkIn(_temp0);
        return super._mgetNoOfPatients(_ppatient.getMh().checkOut(super.getJPACNameDynamic()));
    }
    public _cAZTdata(edu.utulsa.mcs.jpacengine.Certificate _temp0) {
        super(_ppatient.getMh().checkOut(super.getJPACNameDynamic()));
        _ppatient.getMh().checkIn(_temp0);
    }
    static public String getJPACNameStatic() {
        return _ppatient.getJPACName();
    }
    public String getJPACNameDynamic() {
        return _ppatient.getJPACName();
    }
}
```

In Java, fields with the same name can hide one another without causing compilation errors. Hidden fields can still be accessed with proper qualification or type casting (for static and instance fields, respectively). Overridden methods, on the other hand, cannot be accessed. JPAC accessor method names are chosen to reflect not only the name of the field, but also the class (or interface) that contains it. When field accesses are found in JPAC code, the pre-
processor determines the type where the field is stored and generates the appropriate call as needed. An example of this transformation is shown in Figure 11 for the field AZT-data::projectCode.

Unfortunately, generating accessor methods for fields prohibits a simple translation of operators with side effects such as pre/post decrement and increment, and assignment operators (e.g., +=, -=, *=, /= and ^=). However, these operations are translated to functionally equivalent expressions using the accessor methods. Figure 9 shows one such transformation, where the expression noOfPatients++ is replaced by two method calls.

### 3.3. Interfaces and compatibility

Java interfaces help programmers simulate multiple inheritance by providing a location to declare common methods and constants sharable by a number of classes without the use of a common superclass. Although interfaces contain method declarations, these methods are implicitly abstract and must be implemented separately in every class that implements the interface. Fields in interfaces are used as static constants for programmer convenience.

JPAC interfaces are translated in a similar manner as classes, except that interface fields are implicitly static, public and final; this makes them impossible to “privatize.” To complicate matters, Java interfaces do not contain method implementations to permit the addition of accessor methods. JPAC moves interface fields into specially constructed classes making it possible to change their access level to private and to add accessor methods. Wherever a JPAC interface field is referenced, a call to the appropriate accessor method is substituted. The naming of interface fields and accessor methods is performed in the same manner as class fields.

JPAC protected classes are inaccessible from classes not translated by the JPAC preprocessor. To integrate existing Java code and facilitate API interaction, JPAC allows the specification of unprotected packages. Classes and interfaces in packages marked unsecured are not protected by JPAC. Classes and instances in such packages do not adopt the additional certificate parameter for methods or accessor methods for fields. The only modifications are the addition of a package class and the transformation of calls to methods and fields in classes belonging to protected packages.

### 4. Comparison with other work

The Java Development Kit (JDK 1.2) authorizes signed applets based on principal identity and a system-wide policy [9,10]. A protection policy is embodied by the java.security.Policy class [11]. Permissions are represented by the class java.security.Permission, which is inherited by all permissions. The architecture relies on the java.security.ProtectionDomain class to represent principals. Protection domains map principals (defined as a class origin and code signature tuple) to permission sets, and are created as classes loaded at runtime. A system policy file, created by a developer or administrator, defines these mappings. JPAC further extends this protection scheme by adding discretionary access control on a per-package basis. This approach, coupled with the ticket based authorization model, increases the expressiveness
and the security functionality of Java.

Wallach et al. [25] discuss alternate security models for Java based on sophisticated architectures employing capabilities and namespace management. Capabilities are supported by Java’s type safety features that ensure that object references cannot be forged. Namespace management controls how names in a program are resolved into runtime classes to enforce a particular security policy. Standard Java implements a rudimentary version of namespace management, where new classes are loaded dynamically and method calls linked to other classes generate separate namespaces. This mechanism is exploited for system security by recognizing that if some instance or class should be denied access to a resource, then appropriate method calls can be linked to another class with restrictions in place. However, this approach, unlike JPAC, is not well-suited to static (i.e., compile-time) security checking.

Static analysis of security properties is a promising approach. Denning and Denning [5] first proposed a compile-time mechanism for certifying information flow policies. Andrews and Reitman [1] provided a flow logic to verify that programs satisfy confinement properties. Drawing motivation from these approaches, JPAC offers developers the ability to specify information flow (rather than just verifying it) and relies on the standard Java compiler and run-time specifications to signal errors and throw exceptions whenever the established flow policy is violated.

Volpano et al. [24] recast the information flow analysis model in [5] within a type system to establish its soundness. Variables are classified at high or low security levels and information is not allowed to flow between levels. This work led to a sound type system for information flow in a multi-threaded language [19]. Careful inspection of indirect flow of information from high to low variables is performed to detect potential covert channels. Rather than concentrating on one access control model (e.g., MAC), JPAC is built from the ground-up using general ticket based primitives. Thus, it is possible to use JPAC in a variety of scenarios, including DAC, MAC, RBAC and TBAC.

Van Doorn et al. [23] developed secure network objects (SNOs) for Modula-3 that promote sub-typing as a means for specifying security properties for objects. Capabilities and access control lists are used to offer developers the opportunity to express various security policies. JPAC adopts a similar approach, except that it employs (more primitive) tickets instead of capabilities.

Myers and Liskov [18] describe a decentralized information flow control model. The model supports fine-grained access control and static analysis to certify information flow. Security label annotations can be inferred and type-checked by a compiler to verify program information flow properties. Myers implemented these ideas in JFlow [17], a variant of Java that integrates statically checked information flow annotations with advanced programming language features such as objects, subclassing and exceptions. JPAC on the other hand, does not burden the programmers with the task of annotating code with flow information. JPAC also prevents invalid information flow by signaling compile-time errors throwing run-time exceptions.

The SLam calculus is a typed λ-calculus that captures security information of programming elements [14]. A special compiler executes static checks to enforce type system rules guaranteeing program security. Types in SLam are monomorphic and static, but the system is extensible to concurrent and imperative programming. Although important from a conceptual point of view, monomorphism and static typing severely limit its potential. By using Java as the underlying language, JPAC inherits polymorphism, dynamic naming and other object-oriented features. Thus, JPAC strikes a balance between theory and practice.

5. Conclusions

Protection models in existing programming languages and APIs do not provide the expressiveness required for fine-grained, configurable security policies. The JPAC scheme described in this paper extends the Java language to provide programmable access control services for Java packages, including DAC, MAC, RBAC and TBAC. JPAC classes and interfaces can be seamlessly integrated within native Java applications, giving software developers the ability to customize protection policies while reducing the costs associated with such changes. JPAC also offers a solid semantic foundation for the design and implementation of fine-grained authorization models for class-based and instance-based protection. Furthermore, the design makes it possible to statically check software components for security violations using elements of the Java architecture.

References


