Supporting Real World Security Models in Java

Ian Welch and Robert Stroud
University of Newcastle upon Tyne
{I.S.Welch, R.J.Stroud}@ncl.ac.uk

Abstract

The Java Security Model has undergone considerable evolution since its initial implementation. However, due to its historical focus on securing machines against attack from hostile Java applications, it has neglected support for securing "Real World" applications. We suggest that in order to support "Real World" security the ability to insert checks into compiled code in a principled way and high-level abstract security models are required..

We briefly review the evolution of the Java Security Model, outline the requirements for supporting "Real World" security for applications, discuss whether Enterprise Java Beans satisfy these requirements, introduce our approach to meeting these requirements and discuss our current work.

1. Introduction

The Java security model is based on controlling access to abstract resources. However, such a model is inappropriate for real world commercial applications where the security requirements are more concerned with integrity and availability. Furthermore, the Java model requires components to be made security aware at build time and this makes it difficult if not impossible to adapt third-party components to meet additional commercial security requirements. Although it has been suggested that using the Enterprise JavaBeans framework addresses these problems we argue that its wrapper approach is not sufficient to support all real world policies. Instead, we propose that reflection be used as a mechanism to customise components at load-time and that higher-level security models be implemented using a metalevel architecture.

2. Java security model

Java security has undergone considerable evolution. In the JDK1.0 security model [5] any code run locally had full access to system resources while dynamically loaded code had access to system resources controlled by a security manager. The default security manager sandbox provided minimal access and in order to support a different security model a new security manager would have to be implemented. The concept of trusted dynamically loaded code was introduced in JDK1.1 [6]. Any dynamically loaded code that was digitally signed by a trusted code provider could execute with the same permissions as local code. Recently JDK1.2 [7][8] has introduced an extensible access control scheme that applies both to local code and dynamically loaded code. Fine-grained access to system resources by code can be specified in a policy file on the basis of the source of the code, the code provider, and the user of the code. Unlike earlier versions of the JDK this policy file allows the security model to be adjusted without writing a new security manager. This is because the security manager has standard access control checkpoints embedded in its code whose behaviour is determined by the selection of permissions enabled in the policy file. New permissions can be defined but explicit checks for the permissions must be added to the security manager or application code if the permissions apply to application resources rather than system resources.

For example, in order to implement control over access to the payroll operations of a Java payroll application two steps would be carried out. First a `PayrollPermission` class would be defined that extended the basic Java `Permission` class, and encapsulated the security policy. Then the payroll application code would be modified and explicit calls to the `checkPermission` method of an instance of `PayrollPermission` inserted before those operations under the control of the security policy.

This approach suffers from two problems. Firstly, the security policy has to be coded from scratch as no high-level abstractions are provided in Java. Secondly, explicit security checks should not need to be inserted by hand into application code as this is both error-prone and tangles security code with application code making the overall system more difficult to maintain and validate.

In summary, although the security model has become finer grained and more extensible it has remained an access control model with the focus being on the access of downloaded code to system resources. Recent additions to the model have allowed some limited extensibility although in order to support higher-level application security models re-coding of application code is required.

3. Requirements for real world security

Real world security requires richer and more abstract security models than currently provided by Java. It also requires that components can be made secure even if they are not security aware. This allows components to be developed separately from their eventual execution context and provides a separation of concerns between application level concerns and non-functional concerns.

There are two good examples of real world security models that cannot be easily supported by the current Java security model. The first is a resource consumption security model. The second is the Clark-Wilson [4] integrity security model.

3.1 Resource consumption security model

The current Java security model can deny code access to a system resource but it cannot place limits on resource consumption. This leaves a system open to a denial of service attack through as simple an attack as spawning an infinite number of threads. Basically the current model is “either/or” in its application when what is needed instead is a resource consumption model.

3.2 Clark-Wilson security model

The Clark-Wilson model is based on an analysis of security models actually applied within businesses. These models aim at ensuring the integrity of resources rather than simply controlling access to them. They depend on controlling state transformations, and upon maintaining separation of duties between users of the system. The current security model fails in that it would require each controlled operation to be re-coded to include a permission check. This is not appropriate for a component that is delivered in binary form. It would also require determination of which operations update the state of an object so that only those operations that maintain the integrity of the system would be allowed. This approach is both tedious and error prone; what would be better is if all state accesses could be intercepted and only those made from operations that maintain integrity were allowed and all others were blocked.

4. Explicit customisation of the runtime environment

One approach to addressing the resource consumption issue is to add hooks for resource monitoring into the Java runtime environment. Resource consumption policies can then be specified and enforced by the runtime system. The JRes [2] system is an example of this approach. JRes introduces hooks into system classes that provide control over resource consumption.

The main drawbacks to this approach is its lack of generality, and lack of portability. All the hooks are added explicitly to each system class rather than being automatically added under the control of an abstract model. This makes extension and maintenance difficult. Also it requires the use of a customised Java runtime environment making it a non-portable solution. What is required is a general, transparent and portable solution.

5. Enterprise JavaBeans – not a solution

Enterprise JavaBeans (EJB) [1], a component and container architecture, is proposed as a way of extending the Java security model and implementing it transparently. It is envisaged that high-level security models are enforced by the container that the components execute within. The container execution environment could have predefined hooks within it that can be customised in order to manage access to system resources. This would allow a resource consumption security model to be implemented. In addition the EJB architecture allows the interception of all method calls sent to a component through the automatic generation of wrappers. This is seen as an appropriate place to add security controls such as provided by the CORBA security service.

In effect EJB offers a generic architecture for adding in new security behaviours. This at first sight appears to address the problems that arise from explicit modification of system or application classes. However, there are two major problems. Firstly, some security models require not only control over incoming invocations but also the sending of invocations and state update. An example would be a model for confidentiality of communication that required the encryption of all calls to and from a component. Secondly, the CORBA security service does not support high-level security models such as Clark-Wilson.

6. Our approach

Our approach is based on the use of metaobject protocols to provide flexible fine-grained control over the execution of components and the development of a metalevel security architecture that supports security models such as Clark-Wilson and resource consumption. This effectively allows security checks to be inserted directly into compiled code thus avoiding the need to recode applications in order to add application specific security checks.

Figure 1 below presents the overall reflective security architecture. We discuss each aspect of the architecture in the following sections.

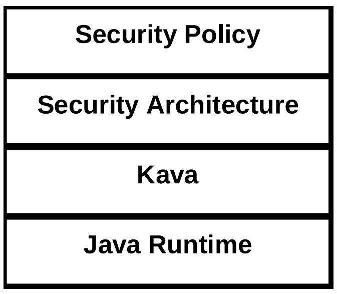

Figure 1. Reflective security architecture.

6.1 Kava metaobject protocol

We have developed a reflective Java called Kava [9] that gives the control over the behaviour of components that is required to support real world security models. It uses bytecode transformations to make principled changes to a component's binary structure in order to provide a metaobject protocol that brings component execution under the control of a metalayer. These changes are applied at the time that components are loaded into a container for execution. The metalayer is written using standard Java classes and specifies adaptations to the behaviour of the components in a reusable way. Although neither bytecode transformation nor metaobject protocols are new ideas, our contribution has been to combine them. Bytecode transformation is a very powerful tool but it is in general difficult to use, as it requires a deep knowledge of class file structure and bytecode programming. What we do is provide a load-time structural metaobject protocol that is used to implement a runtime metaobject protocol. In a sense it is an extension of work done on compile time metaobject protocols. Working at the bytecode level allows control over a wider range of behaviour than is possible with a wrapper approach. For example, the sending of invocations, initialisation, finalisation, and state update are all under the control of Kava.

6.2 Metalevel security architecture

The metalevel security architecture is built on top of the runtime metaobject protocol. It provides an architecture where various security models are unified in an object-oriented model. There is a binding between security metaobjects that co-operate at the metalevel to implement a specific security model and application components. Kava implements the binding by adding hooks directly into the binary code of the components. As this binding exists within the component itself instead of in a separate wrapper class, as with the EJB approach, we argue that we are achieving a strong encapsulation of components. Outside parties cannot bypass the wrapping and therefore the security implemented in the metalevel by simply gaining an uncontrolled reference to the component.

The Kava system, binding specification and the metaobjects must form part of the trusted computing base. The Kava system and binding specification are installed locally and can be secured in the same way as the Java runtime system. However, the metaobjects may exist either locally or be retrieved across the network. This raises the possibility that the metaobjects themselves might be compromised. In order to counter this threat we use a secure version of a classloader that verifies the identity and integrity of the metaobject classes using digital signing techniques. Each metaobject has a digital signature that is encrypted using a private key of the provider of the metaobject. The public key of the provider exists in the local public key registry on the host where the Kava system is installed. The encrypted signature is decrypted and compared with a locally computed signature of the received metaobject. If there is discrepancy then a security exception is raised and the system halts. This prevents malicious combinations of application components and metaobjects.

6.3 Using the architecture – an example

The user of our system does not deal with the implementation details of Kava, they only deal with the high-level metalevel security architecture. For example, the resource consumption model shown in figure 2 is represented in terms of resources, operations, constraints, principals and classes (which may or may not be JavaBeans). In this model a contract is specified that places limits of the invocation of operations on particular resources by applications. The sum of contracts represents a particular resource control security policy. This security policy is then interpreted and implemented through the interception of operations on specified resources. As these include the trapping of the sending of invocations the EJB wrapper approach could not achieve a similar end.

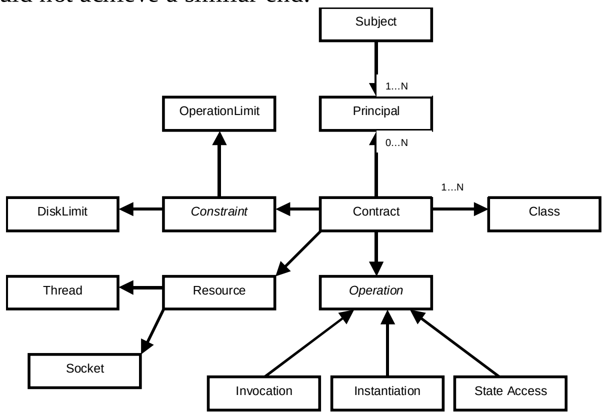

Figure 2. Metalevel Architecture for Resource Control.

An example policy could be to place an upper limit on the creation of threads by an application. This would involve instantiating a `Contract` object that links a `Principal` to a specific set of `Class` objects that represent the application. This `Contract` object also links the `Thread` class to an `OperationLimit` for the `Instantiation` operation. This is then implemented by invoking the `checkConstraint` method of the `OperationLimit` every time a `Thread` is instantiated by the application that is being used by the specified `Principal`.

The benefits of a metalevel security architecture that is developed and configured separately from the application components should be easier maintenance, reduced validation effort and increased reuse. Although the binding step will require some application knowledge, it makes interactions between security and application functions explicit and allows them to be adjusted incrementally and safely at a very late stage in the development of the entire system.

Our approach to modelling security models in an object-oriented way is influenced by the approach taken in the Cherubim project [3]. However, our system is aimed at component security rather than mobile code and supports a greater range of security models. Also the use of bytecode transformation in order to bind the security architecture to the application components differs from their approach which is to define a specialised execution platform.

7. Current work

We are currently evaluating our approach by applying Kava on a case study. Kava is being used to secure a third-party COTS application where only binary components are available. The security meta architecture is being used to implement high-level security models such as Clark-Wilson, MAC [10] and resource consumption. We hope to demonstrate that the application of our approach leads to systems that are easier to extend and maintain due to the good separation of concerns enforced by reflection, simpler to validate due to the reuse of validated security mechanisms in the form of metaobjects, and provide secure implementations of a number of real world security models.

8. Acknowledgements

We would like to thank DERA for their financial support for this work.

9. References

[1] C. Crenshaw, “Developer’s Guide to Understanding Enterprise JavaBeans”, Nova Laboratories, http://java.sun.com.

[2] G. Czajkowski, and T. von Eicken, “JRes: a resource accounting interface for Java”, ACM SIGPLAN Notices Vol. 33, No. 10 (Oct. 1998), OOPSLA '98. Proceedings of the conference on Object-oriented programming, systems, languages, and applications, 1998.

[3] R. Campbell, and T. Qian, “Dynamic Agent-based Security Architecture for Mobile Computers”, the Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98), Australia, Dec 1998.

[4] David D. Clark, and David R. Wilson, “A Comparison of Commercial and Military Computer Security Policies”, IEEE Symposium on Security and Privacy, 1987.

[5] James Gosling, Frank Yellin, and the Java Team, “Java™ API Documentation Version 1.0.2”, Sun Microsystems, Inc., 1996.

[6] Java Team, “JDK™ 1.1.8 Documentation”, Sun Microsystems, Inc., 1996-1999.

[7] Java Team, “Java™ 2 SDK Documentation”, Sun Microsystems, Inc., 1996-1999.

[8] Java Security Team, “Java™ Authentication and Authorization Service”, Sun Microsystems, Inc., http://java.sun.com/security/jaas/index.html, 1999.

[9] I. Welch and R.J. Stroud, “From Dalang to Kava - Evolution of a Reflective Extension for Java”, Meta level architectures and reflection : second international conference, Reflection'99, Saint-Malo, France, 1999.

[10] David E. Bell and Leonard La Padula, “Secure Computer System: Unified Exposition and Multics Interpretation”, ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA 01731 (1975).