13 ROLE BASED SECURITY AND JAVA

D. Smarkusky, S. Demurjian, Sr.*, M. Bastarrica, and T. C. Ting

Abstract: In the past two years, Java has exploded onto the computing landscape, offering an object-oriented language and environment that is suitable for a wide variety of application domains. Java is targeted for applications that include: advanced capabilities in WWW browsers via applets; enterprise computing with database connectivity, CORBA, and RMI; usage in personal, commercial, and consumer market products; embedded computing applications with real-time constraints; and, smart card technology. Security is an integral component of many of these applications, to control access and prevent misuse. The purpose of this chapter is to focus on the security capabilities and potentials of Java. There must be an understanding of the available security primitives in Java, an investigation of the ability of Java to support existing object-oriented security approaches, and a consideration of potential security solutions for distributed object computing applications.

13.1 INTRODUCTION

The Java object-oriented programming language and environment first appeared commercially in early 1996, and in just over 2 years time, there has been an explosive interest and growth of Java across the computing landscape. Java is utilized for distributed, Internet-based applications of all types, including: Web browsers, graphical user interfaces (GUIs), programming environments, mixed-programming language applications, upgrading and interfacing to legacy

*The work of S. Demurjian and C. Bastarrica has been supported, in part, by a contract from the Mitre Corporation, Eatontown, NJ.

S. Jajodia (ed.), *Database Security XII*

systems, etc. Java is also is attractive for general purpose, single-CPU development, since it has the potential to easily evolve software to multiple and varied hardware/OS platforms.

¿From a security perspective, the usage of Java for the design and development of large-scale, multi-processor, distributed applications, is of paramount concern. Successful distributed object computing (DOC) can be addressed from three perspectives. First, when developing new applications, it is often the case that multiple programming languages and varied paradigms must work together, e.g., a Java GUI, a C I/O package, and an SQL database system. This motivates the second perspective, involving the integration of Java with commercial-off-the-shelf (COTS) systems. Of course, when integration occurs, it may be necessary to be innovative and creative to allow interactions with legacy applications, which is the third perspective. In all three perspectives, security must be considered, to insure that interoperating legacy, COTS, and database systems can satisfy the security policy of distributed applications. There have been efforts to begin to address security for DOC [4, 9].

The purpose of this chapter is to examine and detail the security capabilities and potentials of the Java language/environment. Java provides a robust set of security capabilities as part of the `Java Security` Application Programmers Interface (API). These capabilities include digital signatures, message digests, key management, and access control lists. A first goal of this chapter is to detail these capabilities, so that the security community can understand the available functionality. Java, as an object-oriented programming language, is of interest from a user role-based security (URBS) perspective, to determine the potential to realize discretionary access control (DAC). Many researchers, including ourselves, have studied this problem for object-oriented/C++ applications and systems [1, 2, 5, 7]. A second goal of this chapter is to consider the applicability of our previous approaches to Java. This leads to a third goal, an exploration of the unique features of Java that can be used to enhance existing URBS/DAC/OO approaches or to support new approaches for distributed object computing security.

To meet these goals, the remainder of this chapter is organized into four sections and a conclusion. In Section 2, a brief overview of the Java language/environment is provided. In Section 3, the security capabilities in the `Java Security` API, are examined, targeting the first goal described previously. Section 4 explores the realization of URBS/DAC approaches in Java, from prior work [2], targeting the second goal. Section 5 examines advanced security capabilities, concentrating on the potentials of Java, and targeting the third goal. Finally, Section 6 concludes this chapter and outlines future work.

13.2 BACKGROUND: AN OVERVIEW OF JAVA

Java is a third generation, general-purpose, platform-independent, concurrent, class-based, object-oriented language and associated environment. Java can be used to write special programs called *applets* that can be downloaded from the Internet and displayed/manipulated safely within a Web browser, or to develop

standalone applications, with a wide range of capabilities and functionality. Java has two main components, the Java Development Kit (JDK) and the Java Runtime Environment (JRE). The JDK is a package of programs and support files which is needed to develop Java programs. The JDK contains the command-line driven `javac` Java compiler. The Java Debugger (JDB) is included with the JDK. The JRE used to execute Java applications, and consists of the bytecode interpreter and other files such as the code verifier. Version 1.1.6 of both the JDK [12] and the JRE [14] are available from Sun for Microsoft Windows 95/NT 4.0 and Solaris, with third-party ports [13] to a wide variety of other OSs.

In order to support platform independence, Java provides an environment to oversee the execution of applets and applications, the Java Virtual Machine (JVM). JVM is a program which runs on a particular hardware/OS platform (or 'real' machine) which interprets and executes a Java applet/application that is contained in a `.class` file. The `.class` file contains both executable JVM instructions (called bytecodes), and additional information such as the class structure, method and data member visibility, and superclass information. Since each JVM interprets the same set of bytecodes, true program portability is achieved by implementing JVMs for a wide variety of platforms.

The main modeling capability is the Java *class*, which is similar to a C++ class. Within a Java class, a member (method or variable) can be tagged as private, public, protected, or package (default). A class for prescriptions in a health care application is given below:

```
public class Prescription
  {
    public String Get_Prescription_No(...) { ... }
    public void Set_Prescription_No(...) { ... }
    public String Get_Pharmacist_Name(...) { ... }
    public void Set_Pharmacist_Name(...) { ... }
    public String Get_Medication (...) { ... }
    public void Set_Medication(...) { ... }

    private String prescription_no;
    private String pharmacist_name;
    private String medication;
 }
```

Classes that are related to one another can be grouped together into the package abstraction, to be discussed in Section 5 of this chapter. Inheritance is supported in Java by using the `extends` keyword when declaring a class.

Java, through its public interface capabilities and package concepts also requires a clear definition of the exported portion of all classes/packages, which requires software engineers to specifically enumerate which packages, classes, and/or methods are imported. Thus, like Modula-2 and Ada83 (and of course, Ada95), Java provides a set of *application programming interface (API)* packages. The Java Platform 1.1.6 Core API is available online [10]. Each API

contains a complete description of the package, classes and public methods that can be imported and utilized to develop Java applications.

13.3 JAVA AND SECURITY

Java provides transparent, general, and open security mechanisms which do not require any knowledge or action on the part of the software engineer. The *sandbox* is Java's basic security mechanism, which forces downloaded applets to run in a confined portion of the system, and allows the software engineer to customize a security policy. One result of this approach is that the security policy is hard coded as part of the application, providing little or no flexibility either to modify the policy or to have discretionary access control. The Java language/environment has features that assist in protecting the integrity of the system and preventing several common attacks. This section describes the security capabilities in the Java Security API [11].

Sandboxes: An applet's actions are restricted to its *sandbox*, a dedicated area of the Web browser. The applet may do anything it wants within its sandbox, but cannot read or alter any data outside it. The sandbox model supports the running of untrusted code in a trusted environment so that if a user accidentally imports a hostile applet, that applet cannot damage the local machine. To implement sandboxes, the Java platform relies on three major components: the class loader, the bytecode verifier, and the security manager. Each component plays a key role in maintaining the integrity of the system, assuring that only the correct classes are loaded, that the classes are in the correct format, and that untrusted classes will neither execute dangerous instructions nor access protected system resources. Java's *Protected Domains* constitute an extension of the sandbox, and determine the domain and scope in which an applet can execute. Two different protected domains can interact only through trusted code, or by explicit consent of both parties.

The *class loader* determines how and when applets can load classes, and is responsible for: fetching the applet's code from the remote machine, creating and enforcing a namespace hierarchy, and preventing applets from invoking methods that are part of the system's class loader. An executing Java environment permits multiple class loaders, each with its own namespace, to be simultaneously active. Namespaces allow the JVM to group classes based on where they originated (e.g., local or remote). Java applications are free to create their own class loaders. In fact, the JDK provides a template for a class loader to facilitate customization. Before a class loader may permit a given applet to execute, its code must be checked by the *bytecode verifier*. The verifier insures that the applet's code, which may not have been generated by a Java compiler, adheres to all of the rules of the language. In fact, in order to do its job, the verifier assumes that all code is meant to crash or penetrate the system's security measures. Using a bytecode verifier means that Java validates all untrusted code before permitting execution within a namespace. Thus, names-

paces insure that one applet cannot affect the rest of the runtime environment, and code verification insures that an applet cannot violate its own namespace.

Security Managers: The *security manager* enforces the boundaries around the sandbox by implementing and imposing the security policy for applications. All classes in Java must ask the security manager for permission to perform certain operations. `SecurityManager` is an abstract class of the `java.lang` API, and provides the programming interface and partial implementation for all Java security managers. By default, an application has no security manager, so all operations are allowed. But, if there is a security manager, all operations are disallowed by default. Existing browsers and applet viewers create their own security managers when starting up.

When there is a security manager, each operation or group of operations will have its own `checkXXX` method. There are `checkXXX` methods for operations on sockets, threads, files, networking, windows, etc. To write a security manager, it is necessary to create a subclass of `SecurityManager` and override most or all of its methods: `class MyPolicy extends SecurityManager { ... }` . Once a new security manager is created, it can be installed with the `setSecurityManager` method from the `System` class. The security manager will remain active until the end of the application. A method that opens a file for reading invokes the `checkRead` method of the security manager. A method that opens a file for writing invokes the `checkWrite` method. If the security manager approves the operation, the `checkXXX` method returns, otherwise, it throws a `SecurityException`.

Digital Signatures and JAR files: If a particular publisher is trusted, and a signed applet from that publisher has arrived over the Internet and been authenticated, then the Java Security Manager could allow that applet out of the sandbox, and treat it as an application. The first task of any security system is to be able to assure that who or whatever is on the other side of a connection is who or what the user expected to be there, i.e., the host that they have connected to is the host they contacted and not an impostor, or the module that they have loaded is really the one they expected to run and not a substitute. This is of particular concern in downloaded environments where there is a constant threat of a *Trojan Horse.*

The man-in-the-middle/middleman is a type of attack to which all network-based systems might be vulnerable, and proceeds in a number of steps. First, a client application requests some service from a legitimate server. Unknown to both client and server, an attacker application observes this request and waits for the server to respond. When it does, the attacker intercepts the server's response and replaces it with one of its own, one that the client may assume came from the original server. The way to prevent this type of attack is to ship code contained within a digital shrink-wrap, which is achieved in Java using *signed applets.* A supplier bundles Java code (and any related files) into a JAR

(a Java Archive), and then *signs* the file with a digital signature. The client can verify the authenticity of the supplier by verifying the signature.

Key Management: The `Java Security` API provides support for integrated key management in Java programs and applets. Keys are generally obtained through key generators, certificates, or the various `Identity` classes used to manage keys. There are no provisions yet for the parsing of encoded keys and certificates. An identity certificate is a guarantee by a principal that a public key is that of another principal. The `KeyPairGenerator` class is used to generate pairs of public and private keys.

Message Digests: Cryptographers have developed a way to generate a short, unique representation of your message, called a message digest, that can be encrypted and then used as your digital signature. The `MessageDigest` class provides the functionality of a message digest algorithm, such as MD5 or SHA. Message digests are secure one-way hash functions that take arbitrary-sized data and output a fixed-length hash value. Like other algorithm-based classes in the `Java Security` API, `MessageDigest` has two major components: the methods called by applications needing message digest services and the interface implemented by providers that supply specific algorithms.

Access Control Lists: Every authenticated principal will have a level of accessibility: highly trusted resources should be granted more access than those of more dubious origin. Access Control Lists (ACL) are data structures used to guard access to resources, and allow users to define read/write permissions based on users and groups. Each ACL entry contains a set of positive or negative permissions associated with a particular principal (an individual user or a group). Individual permissions (either positive or negative) override the groups' permissions. The `java.security.acl` package provides the interfaces to the ACL and related data structures (ACL entries, groups, permissions, etc.), and the `sun.security.acl` API provides a default implementation.

13.4 JAVA AND USER-ROLE BASED SECURITY

Security issues in distributed object computing are difficult to address since security of individual systems (legacy, COTS, database) must be supported in the distributed and interoperating application. In this section, we consider the ability of Java to support user-role based security (URBS) approaches, where permissions and right to access are be assigned based on the individual roles, rather than to specific users. URBS is a realization of discretionary access control (DAC), that assigns rights and permissions to roles rather than to individual users, with users assigned to specific roles [2, 3].

The premise of our efforts is that the public interface provided by object-oriented programming languages is not suited for the customized approach that is needed for supporting URBS/DAC. The public interface of a class is the union of all privileges (methods) needed by all users of each class. This

allows methods intended for only specific users to be available to all users. Our past approaches have strengthened the public interface concept, promoting the idea that different subsets of the public interface are available to specific users based on role, thereby providing a means to realize URBS/DAC. We have detailed a number of extensible and reusable URBS/DAC enforcement mechanisms that utilize inheritance, generics, and exception handling for the automatic generation of code for DAC policies [2, 3]. This section begins by providing background material on our URBS model. Then, this section reviews the realization of three of our prior URBS approaches in Java. We complete this section with remarks on the limitations of Java in support of our URBS.

13.4.1 User-Role Based Security

To support URBS, the user-role definition hierarchy (URDH) characterizes the different kinds of individuals (and groups) who require different levels of access to an application. In a health care application (HCA) there would be *user roles (UR)* such as `Staff_RN`,`Attending_MD`, `Education`, etc., that can be grouped under a single *user type (UT)* (e.g., `Nurse`). When multiple UTs share privileges, a *user class (UC)* can be defined (e.g., `Medical_Staff`). To define, UCs, UTs, and URs, we utilize a *node profile (NP)*: 1. a name for the node; 2. a prose description of its responsibility; 3. a set of assigned methods (the positive privileges); 4. a set of prohibited methods (the negative privileges); and 5. a set of consistency criteria for relating URDH nodes. Assigned and prohibited methods are of primary interest for our discussion, since they focus on what actions are allowed/denied for each UR.

13.4.2 User-Role Subclassing Approach

In the user-role subclassing approach, URSA, each application class has a group of subclasses, based on the different roles that have some subset of assigned and/or prohibited methods from the class. As subclasses, the basic concept is to inherit and turn off the prohibited methods.

```
public class Prescription { // As given in Section 2 }

public class Staff_RN_Prescription extends Prescription
  { public void Set_Prescription_No(...)
      { return;   // Prohibit access to this method - Turn Off }

    public void Set_Pharmacist_Name(...)
      { return;   // Prohibit access to this method - Turn Off }

    public void Set_Medication(...)
      { return;   // Prohibit access to this method - Turn Off }
  }

public class Attending_MD_Prescription extends Prescription
  { public void Set_Pharmacist_Name(...)
```

```
      { return;   // Prohibit access to this method - Turn Off }
  }
```

If the Set_Prescription_No method is requested by an individual whose role is Staff_RN, then the method associated with the Set_Prescription_No of the Staff_RN_Prescription sublass is executed and no value is returned.

13.4.3 URDH Class Library Approach

In the URDH class library approach (UCLA), a new inheritance hierarchy is used, where each class is a URDH node. For each URDH node, positive methods access is defined based on the assigned methods that have been specified. As the application executes, methods must validate against the current UR.

```
public class Root: All Check Methods defined to return False;
public class Users extends Root {}
public class Medical_Staff extends Root {}

public class Nurse extends Medical_Staff
  { public boolean Check_Prescription_Get_Medication()
      { return True; }
  }

public class Staff_RN extends Nurse
  { public boolean Check_Prescription_Get_Prescription_No()
      { return True; }

    public boolean Check_Prescription_Get_Pharmacist_Name()
      { return True; }
  }
```

The Root class includes new Check methods, which are defined for all application methods to return False. These check methods will be turned on at lower levels (UC/UT/UR) by the assigned methods of the URDH. These Check methods are also utilized to change the code that can be generated for each class:

```
class Prescription extends Item
  { public Prescription(String N, String D, int No, String N1, String M)
      { // initialize variables }
    public int Get_Prescription_No()
      {  if (current_user.Check_Prescription_Get_Prescription_No())
           return (Prescription_No);
         else
           return NULL;
      }
    public void Set_Prescription_No(int No)
      { if (current_user.Check_Prescription_Set_Prescription_No())
           Prescription_No = No;
      }
```

```
    }
```

Once the user role has been determined, a new global `current_user` object will be created at run-time and casted to the selected user role.

13.4.4 Basic Exception Approach

Exception handling in Java is similar to that of C++, where the try construct is utilized to encapsulate a block of code that has the potential to raise an exception. As the code within the try block is executing, various conditions can be checked, and when the correct situation occurs (e.g., unauthorized UR or a call by an authorized UR to a prohibited method), an exception can be thrown and processed by the catch block (e.g., to process the security violation). In the basic exception approach (BEA), each class is modified to include a set to methods for exception handling. This is illustrated below.

```
public class Prescription extends Item {//Private data has been omitted

  public Prescription(String N, String D, int No, String N1, String M)
    { // Assign Prescription variables, call Item constructor }

  public int Get_Prescription_No()
    { return(rtn_int_check_valid_UR(Prescription_No)); }

  // All Other Prescription methods

  public int rtn_int_check_valid_UR(int rtn_int_ck)
    {
      try { Check_UR(); }

      // Catch block to process raised exceptions
      catch (Unauthorized_UR UR_exception)
        { System.out.println("Attempt to access unauthorized UR"); }
    }
  // All other data type check_valid_UR methods

  public void Check_UR() throws Unauthorized_UR
    { if ((compareTo(current_user.Get_User_Role(),"Staff_RN")     != 0)
          (compareTo(current_user.Get_User_Role(),"Attending_MD")!= 0))
        throw new Unauthorized_UR; // throw raises exception
    }
}
```

`Check_UR` is needed to verify that the current UR can invoke the desired method via a table lookup. The class `Unauthorized_UR` is an exception handling class where code can be provided to handle security violations.

13.4.5 Limitations of Java in Support of URBS

While Java appears to easily support the various approaches given in Sections 4.2, 4.3, and 4.4, in actuality, Java has some limitations:

- UCLA, as presented in Section 4.3, is in fact, not fully supported. UCLA as originally conceptualized in C++ [2] requires multiple inheritance, which is needed to definite the URDH. While Java can realize UCLA through the replication of privileges from User into either the user types or user classes, it is not an ideal solution. The interface capability of Java, which supports design-level multiple inheritance, is also not appropriate, since interfaces do not allow implementations to be inherited.
- UCLA and BEA, as presented in Sections 4.3 and 4.4, respectively, have corresponding approaches (GUCLA and GEA) that utilize generics [3]. Java, without generics, the ability to reuse security definition and enforcement code is a major drawback of the language.

While Java appears to have stabilized from a language design perspective, the user community may call of the inclusion of both multiple inheritance and generics, since both concepts are fundamental to software reuse.

13.5 ADVANCED SECURITY FEATURES AND URBS

This section focuses on the third goal of the chapter, the ability to utilize security features of Java for URBS, thereby truly exploring the potentials of the language. The remainder of this section considers four advanced capabilities of Java and their potential for supporting URBS: packages for encapsulating security definition and enforcement code; access control lists; the `Class` class of the Java Language API; and, software agents which are supported by Java aglets.

Packages in Java The highest level of abstraction/encapsulation in Java is the package, which allows collections of one or more classes to be bound into a single named unit. For example, consider a `PatientInfo` package:

```
package PatientInfo;
  class Prescription { ... };
  class PatientGUI { ... };
  class MedicalRecord { ... };
        ...
```

```
package PatientInfo;
  public class Prescription { ... };
  public class PatientGUI { ... };
  public class MedicalRecord { ... };
            ...
```

In the version on the left, the classes are only visible within the package in which they are defined. In the version on the right, classes tagged with the public qualifier are visible within the package and externally.

The package construct can be instrumental in encapsulating the security definition and enforcement code that is required for the different URBS approaches. In UCLA (see Section 4.3 again), the entire URDH class library can be encapsulated into a single package, allowing changes to the URDH to be localized to

a single, controlled package. In URSA (see Section 4.2 again), `Prescription`, `Staff_RN_Prescription`, and `Attending_MD_Prescription` can be encapsulated into a single package, with `Prescription` not tagged as a `public` class. This would mean that only the other two user-role subclasses, tagged as public, are visible externally, which would further protect unauthorized access to `Prescription`, since all access must go through the user-role subclasses.

Access Control Lists The main purpose of the URDH is to allow methods that are defined on classes throughout the application to be assigned and/or prohibited to various user classes, user types, and user roles. The privileges associated with the URDH are directly supported in Java via the Access Control List (ACL). Each ACL entry contains a set of permissions (access to methods) and for a particular principal (UR or UT). Privileges are assigned when the principal is allowed to access a method and prohibited otherwise. The individual permissions (for URBS, the UR) will override permissions of the group (for URBS, the UT) to which an individual belongs. The following methods from `java.security.acl.ACL` are required for the support of URBS:

1. **addEntry()**: Adds an ACL entry to the Access Control List. This entry contains the specified user and a list of methods which are assigned or prohibited for this user, according to the user role that is being played.

2. **checkPermission()**: Returns true if the input user has permission to access the input method, false otherwise.

3. **getPermission()**: Returns an enumeration of all methods which are assigned or prohibited for the input user. The assigned/prohibited methods are determined by first obtaining the assigned/prohibited methods for the group (in our case the UT) and then determining the assigned/prohibited methods for the individual (in our case the UR). The final permissions are then determined by allowing the individual permissions to override the group permissions for both the assigned and prohibited methods. The assigned permission set is returned.

The following methods from `java.security.acl.ACLEntry` are required to build a URBS ACL entry:

1. **addPermission()**: Adds a permission (method) to the ACL Entry.

2. **checkPermission()**: Determines if a permission (method) is already part of the ACL Entry.

3. **removePermission()**: Removes a permission (method) from the entry.

4. **setPrincipal()**: Specifies the user role or user class for which the permissions (methods) are assigned or prohibited.

5. **getPrincipal()**: Returns the user role or user class for which these permissions (methods) are assigned or prohibited.

6. **setNegativePermissions()**: Set the ACL entry to be a list of negative permissions (prohibited methods).

For URBS, we would specify the permissions of all methods so that the method `CheckPermission()` could be invoked to accurately determine both the assigned and prohibited methods. As we stated earlier, the URs inherit all of the permissions of the parent UT. The Java API `java.security.acl.Group`, can be used to assign URs to the UTs, via the methods `addMember()` and `removeMember()`, where the UT would be the Group.

For URSA, UCLA, and BEA, ACL can be utilized to track the information required for an authorization list, that would bind users to their associated roles upon login. For BEA, the ACL has the most significant potential use, namely for the `Check_UR` method, that is able to verify which URs have access to which methods. Using an ACL, this information could be dynamically changed, whenever the security requirements cause the addition/deletion of roles or changes in application classes. While an ACL can be implemented in any language, having one designed, implemented, tested, and with a standard interface, is a definite advantage to Java.

The `Class` Class in Java In the `java.lang` API, the `Object` and `Class` classes have a large set of methods defined that are accessible to software engineers for obtaining information about any system- or user-defined class in Java. For instance, `Class` has methods that can be invoked to return, for a specific user or system class, a list of its public methods, member variables, declared constructors, etc.

The `Class` class can be used by URSA, UCLA, and BEA for the dynamic retrieval of all public methods for each class. The retrieved methods would have a default permission of assigned and only the links to the prohibited methods would need to be removed. For example, the `Check_UR` method of BEA, if implemented with ACL as described earlier, could utilize the `getMethods` method of `Class` whenever the security policy was updated. This would allow the revised/updated entries of the ACL (that contain, for each role, the assigned methods) to be automatically and dynamically compared against the actual methods defined on each class. Similarly, whenever a class was altered, this verification could also occur. In both situations, the maintenance of the security policy is greatly simplified.

Java and Aglets Mobile *software agents* are defined in formal terms as objects that have *behavior, state* and *location* [8]. Agents can move from place to place and have a specific function or responsibility to perform. Agents are like other objects in that they can be created and destroyed, but they can also migrate to a new location, execute their required responsibilities, and process incoming messages from other agents. Agents cannot interact by invoking each others methods, rather, they communicate via message passing.

IBM terms these mobile agents of Java, "aglets", combining the terms of "agent" and "applet" [16]. Unlike a Java applet, an aglet continues execution

where it left off (upon reaching a new location). This is possible because an aglet is an executable object (containing code and state data) which moves from host to host across a network [15]. Karjoth describes a *proxy* as a representative of an aglet which serves as a shield to protect the aglet from direct access to its public methods [6]. The proxy used for the aglet has the responsibility to prevent the access of unauthorized users (or agents).

Like applets, aglet actions should be restricted to a sandbox (see Section 3 again). The sandbox model supports the running of untrusted code in a trusted environment so that if a hostile aglet is received, that aglet cannot damage the local machine. For applets, this security is enforced through three major components: the class loader, the bytecode verifier, and the security manager. Aglets would require the same level of security as applets. The aglets would need to ask permission from the security manager before performing operations, thus allowing the security manager to know the identity of the aglet.

Mobile aglet security is progressing with the use of the Java sandbox mechanism and separation execution environments [6]. Java security mechanisms such as cryptography and authentication are also be investigated to ensure security of both the aglet and the messages transported between aglets. Aglets offer the opportunity to rethink our URBS security approaches, which are class/method based for user roles, and whose definition process is focused on type-level concerns. In distributed object computing, it is critical to explore the security of runtime objects, as they are accessed by users playing roles. Aglets may provide active objects that monitor and/or enforce security, from the perspective of the user, the user role, the object, or any/all combinations. As security needs change, security aglets can be dynamically updated to maintain their oversight and enforcement capability. Aglets in Java must be examined for their potential to support security in distributed object computing.

13.6 CONCLUDING REMARKS AND FUTURE WORK

This chapter has examined the security capabilities and potentials of the Java object-oriented language/environment. There are a wide range of security capabilities, provided in the `Java Security` API, including digital signatures, message digests, key management, and access control lists, which all function under the control of a security manager, as described in Section 3. From an object-oriented/programming language perspective, Section 4 examined the ability of Java to support our previous URBS approaches. While some of the approaches were realizable in Java, others that utilize multiple inheritance and generics could not be fully attained. Section 5 examined advanced security capabilities of Java in support of URBS. Specifically, we considered the package abstraction for encapsulating URBS code, Java's access control lists for realizing important components of our URBS approaches, and the `Class` class for performing automatic and dynamic verification of security privileges.

One of the more interesting potentials of Java is related to our future work, namely, the utilization of Java agents, or aglets, for supporting security in distributed object computing. Another future related area is the support of

security within the CORBA/ORB framework, which is the only available standard for distributed computing. A third related area is security capabilities offered by emerging object-oriented database platforms, including the recently announced Jasmine by CAI.

References

[1] J. Barkley, "Implementing Role-Based Access Control Using Object Technology", *Proc. of First ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, November 1995.

[2] S. Demurjian, T. Daggett, T.C. Ting, and M.-Y. Hu, "URBS Enforcement Mechanisms for Object-Oriented Systems and Applications", in *Database Security, IX: Status and Prospects*, D. Spooner, S. Demurjian, and J. Dobson (eds.), Chapman Hall, 1995.

[3] S. Demurjian, T.C. Ting, M. Price, and M.-Y. Hu, "Extensible and Reusable Role-Based Object-Oriented Security", in *Database Security, X: Status and Prospects*, D. Spooner, P. Samarati, and R. Sandhu (eds.), Chapman Hall, 1997.

[4] S. Demurjian, T.C. Ting, and J. Reisner, "Software Architectural Alternatives for User Role-Based Security Policies " *Database Security, XI, Status and Prospects*, T. Y. Lin and X. Qian (eds.), Chapman Hall, 1998.

[5] W. Gotthard, et al., "System-Guided View Integration for Object-Oriented Databases", *IEEE Trans. on Knowledge and Data Engineering,* Vol. 4, No. 1, Feb. 1992.

[6] G. Karjoth, D. Lange, M. Oshima, "A Security Model for Aglets", *IEEE Internet Computing*, Vol. 1, No. 4, July-August 1997.

[7] F. Rabitti, et al., "A Model of Authorization for Next Generation Database Systems", *ACM Trans. on Database Systems*, Vol. 16, No. 1, March 1991.

[8] B. Sommers, "Agents: Not just for Bond Anymore": `www.javaworld.com/javaworld/jw-04-1997/jw-04-agents.html`

[9] Z. Tari, "Designing Security Agents for DOK Federated System", in *Database Security, XI, Status and Prospects*, T. Y. Lin and X. Qian (eds.), Chapman Hall, 1998.

[10] Java Platform 1.1.6 Core API Specification: `www.javasoft.com/products/jdk/1.1/docs/api/packages.html`

[11] Javasoft's API Documentation - Package java.security: `www.javasoft.com/products/jdk/1.1/docs/api/Package-java.security.html`

[12] The Java Development Kit(JDK): `java.sun.com/products/jdk/1.1/index.html`

[13] Operating Systems Supporting Java: `java.sun.com/products/jdk/jdk-ports.html`

[14] Downloading the Java Runtime Environment: `java.sun.com/products/jdk/1.1/jre/index.html`

[15] "Under the Hood: The architecture of aglets", JavaWorld, April 1997: www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html

[16] IBM Aglets Workbench Home Page: www.trl.ibm.co.jp/aglets/