Using Self-Defending Objects to Develop Security Aware Applications in Java™

John W. Holford[1], William J. Caelli[2], Anthony W. Rhodes[2]

Information Security Research Centre[1],School of Software Engineering and Data Communications[2]
Queensland University of Technology,
Brisbane, Queensland 4001,
Email: j.holford@qut.edu.au

Abstract

The self defending object (SDO) approach to the development of security aware applications represents a change in the object oriented paradigm, whereby the software objects that encapsulate sensitive data or provide security sensitive functionality are responsible for its protection. Such an approach aims at defining and testing new concepts related to the growing requirements for information assurance in information systems. It involves a shift in the way in which application developers look at objects. Rather than acting as containers and dispensers of data, software objects become actively responsible for the protection of that data. By basing the design of security aware applications on the SDO concept, the provision of application specific, user centric, access control is simplified. When using the SDO approach, the access control mechanisms are localized within those objects that encapsulate sensitive data and functionality rather than being distributed throughout the application. Consequently, security measures are consistently applied and are not bypassable.

The major contribution of this paper is to discuss how the SDO concept that was introduced in (Holford, Caelli & Rhodes 2003), can be used in the development of security aware applications. It begins by briefly presenting the rationale behind the SDO concept and its applicability to software design. It continues with a discussion of the experiences gained from using the SDO concept in the development of prototype security aware applications in the Java™language and concludes by outlining future work aimed at extending the concept to the provision of 'self defending' software components and finding solutions for the trusted deployment of such components.

Keywords: Trusted Systems, Access Control, Security Architecture, Computer Security, Information Assurance, Object Oriented System.

1 Introduction

The object oriented (OO) paradigm does not provide the support required for the development of security aware applications (SAAs)[1]. To remedy this shortcoming, an extension to the OO paradigm to incorporate the concept of self defending objects (SDOs) is proposed. A SDO is an enhancement of the 'normal' software object such that it not only encapsulates some sensitive data, but is also aware of the sensitivity of that data and so protects that data and sensitive methods from inappropriate access. Currently such a concept is foreign to the OO paradigm. Not only is the development of SAAs based on the SDO concept simpler, but the security aspects of the application should be more reliable. Currently only the applicability of the SDO concept to the development of non-distributed 'custom built'[2] SAAs is being considered.

In this paper, the SDO concept is introduced and the rationale for its adoption discussed. The features provided by the Java™ development environment that support the SDO paradigm are described and an overview of how application developers might use the SDO approach to develop SAAs is provided. The paper continues by discussing the experiences gained through developing small Java™ applications that were specifically written to prototype the SDO approach. The paper concludes by identifying future work aimed at extending the SDO concept to the development of 'self defending' software components and finding solutions for the trusted deployment of such components.

2 The Rationale for using Self Defending Objects (SDOs)

With the widespread use of C++, Java™ and the .Net languages, the object oriented (OO) paradigm is experiencing a high level of adoption for the development of application programs. Much of its success is due to the intuitive way in which application developers are able to reason about programs and to abstract away much of the complexity inherent in the programming task. When combined with inheritance, which simplifies design and provides a convenient vehicle for code reuse, the success of the object oriented paradigm is of little surprise.

Object oriented design is data-centric and relies on identifying the data present in the system and associating functionality with that data. Classes and their individual instantiations, objects, are responsible for the encapsulation of that data and providing a set of operations appropriate to that data. Good programming practice dictates that the methods of the object provide the only mechanism by which that data may be accessed.

Through the localization of all code with direct access to data within its encapsulating object, the design of software is simplified and the integrity of the applications' data structures is enhanced. The OO paradigm also simplifies program testing and debugging. If the corruption of data encapsulated within an object occurs, that corruption is necessarily caused by an error in the code of that object. Prior to the adoption of the OO approach (and its predecessor, the abstract data type approach), that flawed code could have been located anywhere in the application. Consequently the adoption of the OO paradigm has

[1] A SAA is an application that exhibits behaviour that is based on security information. They often enforce fine-grained or application-specific security policies and react intelligently to the occurrence of security problems.

[2] A custom built application is one that is constructed using those objects provided by the application development environment and those objects written for the application.

resulted in a significant improvement in program reliability.

There is however, a major deficiency with the OO paradigm when applied to the development of SAAs. The current OO approach is primarily concerned with ensuring a method does not violate the integrity of data encapsulated within that object. Consequently when an object receives a message, the request will be satisfied. Whether that action should be permitted is not considered. Within the object oriented paradigm, an object is unaware of the sensitivity of its encapsulated data and functionality, so does not mediate any request to access its encapsulated resources. Such an approach is appropriate for the majority of applications; however it is not appropriate when SAAs are being developed. By their very nature, SAAs take security considerations into account when performing their tasks. It should be expected that those objects within an SDO which encapsulate sensitive information should be security aware, and so take (at least partial) responsibility for defending that information from unauthorized access.

2.1 Why do we need SDOs? - A Scenario

To illustrate how the SDO concept might be applied to the development of SAAs, an overview of the payroll subsystem of a human resources (HR) information system, will be discussed.

Most employees are either full time employees who are paid a fixed yearly salary or casual employees who are paid by the hour. Executives however, may have a range of salary packages depending upon their position within the corporate hierarchy.

In addition to the need to provide the desired functionality of the system, the requirement to maintain the privacy of that payroll information is paramount. Information should only be available to those personnel authorized to view and modify it. A range of privacy requirements would typically apply within such systems. For example, it would be typical for the package details of the CEO and other top executives to be highly restricted, with possibly only the human resources manager being able to view and modify that information. Conversely, the pay details of an administrative assistant might be available to most human resources personnel. The action of the payroll system is dependent upon the privacy/security policy of the organization, i.e. dependent on the user accessing the system and the information being accessed. Such an application would obviously be classified as being security aware.

In a typical OO design, a hierarchy of classes will be created to encapsulate the payroll information and provide the non-privacy related functionality necessary to construct that portion of the HR system. Privacy related issues would normally be handled on a more or less ad hoc basis. Before displaying the salary details of an individual employee, the logic of the main application applies the relevant access control rules to determine whether those details should be made available to that user. Should the current user not possess the privileges needed to access that particular data, say the salary package details of the CEO, that data will not be displayed. Such an approach certainly works as evidenced by the existence of such applications. The major problem with such an approach is that the access control rules must be applied whenever salary details are accessed i.e. from potentially numerous places within the application. Such an ad hoc approach makes the possibility of not applying the necessary access control rules somewhere in the application, unacceptably high. Such an omission could result in the subsequent divulging of sensitive information, say the CEO's salary details to an unauthorized user.

If the SDO concept were applied to the design of that application, the object which encapsulates the CEO's salary details would be responsible for protecting that information. In particular, the methods of the CEO's payroll object (which make that information available) will apply the specific access control rules which apply to the CEO's information. Unless the access is being made by the HR manager, that request will be rejected.

Hopefully the consideration of this simplified example has clarified the philosophy behind the SDO concept, and highlighted its relevance to the development of SAAs.

2.2 What is an SDO?

An SDO is like any other object in that it provides the functionality associated with its encapsulated data. However an SDO encapsulates resources which have been identified within the security requirements for the application as requiring protection and is aware of the need to protect those resources. Consequently, an SDO differs from other objects within a security aware application in two major ways: (1) each of the public methods which provide access to protected resources, is passed an authentication token as a parameter and (2) the preamble of those methods includes an authorization check, based on the supplied authentication token, to ensure that the principal / user has the right to invoke that method.

The control of access to the resources of an SDO is achieved by controlling:

- the instantiation (creation) of SDOs. In a particular context, the ability to instantiate a particular SDO object may expose the application to a potential security breach, so must be controlled.
- access to mutator and accessor methods. The relevant data member is protected from unauthorized access by controlling who has the the ability to invoke the method.
- access to methods which access multiple data members. Placing access control restrictions on the method as a whole is convenient in that it permits a quick check which determines whether that user should be permitted to access multiple data members in the manner performed by the method. However, to achieve more complete protection of all of the data members being accessed, the method authorization alone may be considered insufficient. In such cases, authorization for each access to a data member could be required. Depending on the preference of the development team, such protection could be provided by either accessing the individual data members through their accessor or mutator methods, or alternatively, by providing separate authorization functionality for each data member and the mode of access. (Such an approach would effectively duplicate the accessor / mutator protection logic).
- the access to methods providing sensitive functionality that is unrelated to the encapsulated data. In the context of the scenario discussed in Section 2.1, such functionality might be the determination of whether an applicant for a particular position has the relevant educational qualifications.

The other major aspect of the protection of resources undertaken by an SDO is the generation of that portion of the audit log which relates to its resources. If specified within the application's auditing

requirements, the authorization process will involve the appending of both the result and details of the authorization check to the audit trail. The token passed to the SDO to permit authorization, also must include the identity of the individual to permit the creation of such an audit record, even if the authorization is based on role based access control.

2.3 The SDO Protection Mechanism

An SDO defends its sensitive resources by restricting access to its methods and data members. The access control is based on the security policy for the application and the originator of the request. In response to a message received an SDO, information may be made available to some principals / users but not others, while fewer still would be permitted to modify that data.

Mandatory access control (MAC) has been studied for many years, normally in the context of trusted operating systems, where it has been successfully applied to the protection of sensitive system resources such as files, sockets and processes. Mechanisms based on the use of access control lists or capabilities have ensured that such resources are only accessed in accordance with the system's security policy. SDOs will also apply MAC to their resources. However, the access control provided by SDOs differs from that provided by an operating system. An SDO will be providing fine grained access control aimed at restricting access to individual methods and the data members of the SDO. Such protection is also application specific. In contrast, an operating system provides much more coarsely grained protection and provides protection which is system wide. Despite those differences, the implementation mechanisms employed by trusted operating systems are equally applicable by SDOs.

In trusted operating systems, the management of the MAC policy information has proved to be a major problem. That problem has been overcome to a large extent within trusted operating systems, including Sun's Trusted Solaris, by providing support for an alternative form of MAC, namely role based access control (RBAC). When using RBAC, rather than directly granting privileges to users, users are associated with roles and roles are granted privileges. To demonstrate the advantage of using such an approach, consider the addition of a new user to the computer system. A new user is allocated all the privileges they require by associating them with one or more roles. They no longer need each of the numerous privileges that they require to be separately allocated to them. To simplify the management of the application's security policy information, it is anticipated that for most applications, RBAC rather than classic MAC, will be enforced by SDOs.

The major advantages of performing access control checks within objects (making them SDOs) rather than distributing them throughout the application, are:

1. the methods of an object provide the ideal location for applying access control measures. As these methods provide the only access to that data, applying the access checks at that point ensures the checks cannot be bypassed.

2. the localization of all code that accesses data to the encapsulating object results in an increase in software reliability. The localization of access control measures within the encapsulating software object, is expected to produce an analogous increase in the reliability of the application's access control measures.

3. by distributing the access control checks throughout the SAA code of an SAA, it is difficult to verify that appropriate access control checks are always applied. With the SDO approach, the code performing access control checking is located within the class file of the encapsulating object, so the verification of access control semantics using code reviews is simplified.

Another important aspect of protecting sensitive resources is the maintenance of the application's audit trail. By shifting the responsibility for writing audit logs to the SDOs, the generation of the audit trail is also simplified. Like the access control mechanism itself, the logging of the specified authorization events, will be non bypassable.

In summary, the adoption of the SDO paradigm ensures that the fine grained application specific access control measures required of an SAA are implemented in a manner which ensures that they are both appropriate, non-bypassable and logged. The methods of a self-defending object provide the only access to its encapsulated sensitive data and are also responsible for protecting that data.

3 Java™ Language Support for the SDO Concept

The SDO concept is applicable to the development of SAAs in any of the modern object oriented languages. Java™ has been chosen for the development of the prototype SAAs used to test the viability of the SDO concept. That choice was based not only on the popularity and elegance of the language, but also the presence of several security related features provided by the object library. The features which were of particular importance in the development of the prototypes included security managers, the Java Authentication and Authorization Service Framework (JAAS) and support for object protection. The relevance of these features will be discussed in the remainder of this section.

3.1 Controlling Access to Resources

In the Java™ Development Kit (JDK) 1.2 security architecture, the permissions granted to executing code are specified within a security policy. Permissions granted may be conditional, depending on the origin of the code or the presence of the digital signature of a trusted party. When an object attempts an access to a sensitive resource, such as a file or system property, the security manager ensures that access is permitted by the security policy. Unless a security policy is explicitly specified, the sandbox policy is used. That policy severely restricts the activities of applets while granting applications full access to resources.

The default implementation of the JDK reads the security policy from an ASCII file and instantiates a corresponding policy object for use by the security manager. The overriding principle of the security model is that unless a privilege is explicitly granted, access to the corresponding sensitive resource is denied. The notion of granting privileges other than those specified, is not supported.

3.2 Java Authentication and Authorization Service Framework (JAAS)

Prior to the introduction of the JAAS framework, only a code-centric approach to access control was employed. Such an approach is inappropriate for those SAAs which demand a user-centric approach to access control i.e. the user of the code determines the restrictions placed on that code, not just the origin of the

code. With the incorporation of JAAS within JDK 1.4, support for both code-centric and user-centric approaches to access control is provided.

Central to the JAAS framework are the concepts of a *principal*, a name (possibly one of many) associated with a user, and a *subject*, the collection of principals and security credentials associated with a user. Thus a single 'person' (subject) may have several identities, with a different identity being used to perform different actions.

3.2.1 JAAS Authentication

Under JAAS, an application is independent of the authentication mechanism being used. Such an approach allows different authentication mechanisms[3] or indeed multi-factor authentication, to be plugged into the application as security requirements dictate. Such changes are made without the need to make modifications to the code of the application. To achieve such independence, the application is required to interact with a *LoginContext* rather than with the authentication service or services directly. The login context uses configuration information to determine not only which authentication services are to be employed, but also when multiple services are used, how those services should interact with one another. (An authentication service is referred to as a *LoginModule* within JAAS documentation.)

To support multi-factor authentication, the login context performs user authentication in two stages:

1. each of the login modules is called upon to authenticate the user. When multiple login modules are employed, the configuration information specifies whether the user must successfully authenticate with all, or just a specified subset of the login modules, for the login to be considered successful. The configuration also controls the ability of login modules to share authentication information and so support single sign-on.

2. each of the login modules is instructed to either abort or commit the login. If the login is to be committed, relevant principals and/or credentials are created by each of the services and associated with the subject.

Login modules could be plugged-in under different types of applications, e.g. GUI and non-GUI based, so a mechanism by which the login modules are able to communicate with the user to gather authentication information is required. In JAAS, this is achieved through the use of callbacks. The application provides a *Callback* implementation to the login context which enables the various login modules to interact with the user in an application specific fashion.

3.2.2 JAAS Authorization

The incorporation of JAAS into the Java™ security model necessitated an extension to the security policy to allow permissions also be granted on the basis of the principal running the code. Within the policy file, JAAS user-centric statements may be freely intermingled with code-centric statements.

When an application is executed, the code is granted those permissions allocated on the basis of the origin of the code and/or digital signatures. However, to be granted those permissions associated with a principal, a thread needs to be associated with the relevant *Subject*. That association is made by invoking one of the static methods of the *Subject* class, either *doAs* or *doAsPrivileged*. Thereafter, the actions of that thread are mediated by the security manager on the basis of both the user-centric and code-centric permissions that it now possesses.

The method *doAs* creates a new thread and associates it with a combination of the current access control context and the permissions associated with the principals of the specified subject. When the *doAs* method is used to associate the thread and the subject, Gong (Gong 1999), described the action of the security manager as: *'When a JAAS SecurityManager must make an access control decision, it investigates the stack and retrieves all of the subjects associated with the operations on the stack. It checks and ensures that every subject on the stack is granted the permission.'*

The effect of the *doAsPrivileged* method is similar except that the supplied *AccessControlContext* is used instead of the current access control context and access decisions to be based purely on the current subject (rather than all the subjects associated with operations on the stack). That method call has the effect of temporarily raising the privileges of trusted code to those associated with that subject alone, allowing the thread to perform actions that would otherwise be disallowed. The situation is analogous to making an operating system call. There too, a thread while executing trusted code is granted privileges to which it would not otherwise be entitled.

The security manager or a *AccessControlContext* object directly, can be used to check whether (programmer-defined) user centric permissions have been granted. This checking is achieved by invoking the *checkPermission* method of either class. When the security manager is used to perform the permission check, it determines whether the current thread has the permission being checked. On other hand, the access control context determines whether the access would be permitted in its context, not necessarily, the current access control context. Using the access control context has the advantage of not requiring the current thread to be running in the required security context, for a permission check to be performed.

3.2.3 JAAS and SDOs

An SDO and JAAS provide complementary aspects of SAA development with both concerned with the provision of user-centric access control. The SDO concept provides the philosophy behind the design of the access control measures of the SAA, while JAAS provides the programming language support that enables the implementation of that philosophy. JAAS authentication is particularly useful in that it permits new authentication mechanisms to be plugged into the application and supports the use of multi-factor authentication to provide strong authentication.

JAAS authorization provides useful facilities for implementing the authorization required to implement the SDO philosophy. As previously introduced, two of the mechanisms available in JAAS to permit permission checking are through the use of the security manager and an access control context.

To permit JAAS mechanisms to be used to mediate access to the methods of a SDO, a user-defined permission is created for each SDO method. Following the naming conventions used by Sun for other permissions, the name of the permission is the fully qualified name of the method with which it is associated. Such a permission entitles its holder to the right to invoke the method. The preamble of the protected methods firstly creates the appropriate permission and then performs a permission check, to ensure

[3] Authentication could rely on the direct interaction with the user, the possession of credentials or the use of smartcards or biometric information.

that the required permission was held by the current user.

A similar approach can be used to restrict access to the individual data members of an SDO, namely the creation of a separate permission for each data member and checking for that permission prior to accessing a data member.

SecurityManager Mediated Access Control When using the security manager, the current thread needs to be associated with the required subject. That approach has an advantage of not requiring the explicit presentation of an authentication token to obtain the related privileges. Instead the thread must be executing in the context of the required subject and so will possess the associated privileges.

There are significant limitations with the use of the security manager mediated checks if one wishes to construct GUI based SAAs. In a GUI application, the actions requested by the user are carried out by the event handlers associated with the GUI components. Unfortunately, the event handler does not execute with the privileges of the current user until that system created thread has been associated with the subject. The event handler must spawn yet another thread which is associated with the current subject (using a doAsPrivileged call). That second thread performs the actions that one would have expected the event handler execute. Consequently there is an significant increase in code complexity and a performance penalty.

For GUI objects which have substantial data models containing numerous SDOs, that performance penalty is likely to be prohibitive. Suppose that the GUI incorporates a *JTable* and each of the rows of the JTable displays data which is being extracted from an SDO. Whenever the JTable needs to be re-rendered, a new thread and security context must be created for each cell that extracts data from an SDO. From an efficiency standpoint, this situation is disastrous. Conversely, the more efficient approach of storing only extracted data in the data model of the GUI object, defeats the benefits of using SDOs.

When GUI based SAAs are being developed, the use of the security manager to perform the access permission checking is inappropriate.

AccessControlContext mediated Access Control A *AccessControlContext*[4] (ACC) object provides an alternative way of checking whether a particular permission is held. An ACC object which is associated with the privileges of the current user is included within the authentication token which is passed to the SDOs. The required authorization decision is then made within the SDO object by invoking the *checkPermission* method of the ACC. The GUI event handlers and renderers of the application would also pass an ACC object to the SDOs avoiding the inefficiencies associated with the security manager approach.

The role of JAAS It is believed that JAAS authentication is an extremely useful mechanism of providing the user authentication on which SDO based SAAs rely.

JAAS authorization is potentially very useful when developing SAAs where the ability to access a resource can be determined on the possession of a privilege alone. The use of security manager to mediate access control appears to be limited, particularly when GUI based applications are being developed. On the other hand, JAAS authorization based on the use of the access control context appears to be useful in mediating access to both SDO methods and data member.

[4] A convenient way to obtain an ACC object is by creating a thread running with the required privileges and making an *AccessController.getContext* call.

3.3 Java™ Support for Object Protection

Java™ provides support for object protection by providing mechanisms by which an object may be (1) cryptographically signed to protect its integrity, a *SignedObject* (2) encrypted to provide confidentiality, a *SealedObject* and (3) associated with a guard object which provides access control, a *GuardedObject*.

3.3.1 Signed and Sealed Objects

A *java.security.SignedObject* consists of the object being protected and a cryptographic signature for that object. The protected object can only be extracted from the signed object if its associated digital signature is valid. Like a signed JAR file, a signed object provides a mechanism by which any loss in the integrity of an object is detectable. A signed object differs from a signed JAR file in that it provides a mechanism for the signing runtime objects, rather than just static code, and so provides a mechanism for validating the integrity of both the state of the object as well as its functionality.

Possible uses for signed objects that were identified by Gong (Gong 1999) include the provision of unforgeable authentication and authorization tokens, and the protection of objects being transmitted between virtual machines or those held in storage.

A *java.crypto.SealedObject* provides a mechanism for preserving the confidentiality of an object. A sealed object encrypts the object and then encapsulates the resulting encrypted object. The object can only be recovered by providing the required cryptographic key and cipher to the sealed object.

If a guarantee of both the integrity and confidentiality of an object is required, the object should firstly be signed by creating a signed object, and then encrypted by creating a sealed object from that signed object.

When developing non-distributed SAAs, the use of both signed and sealed authorization tokens although desirable, will prove too expensive on many occasions. If such an approach were to be used, each access to an SDO method would involve the decryption of the presented token and the verification of its digital signature before the 'actual token' would be available to be used to authorize the access. However as SDO authorization is based on the presentation of such tokens, the integrity of such tokens should be of considerable concern. Where the sensitivity of the protected data is such that the violation of the integrity of messages being sent within the virtual machine is of concern, the sending of signed authorization objects may be warranted.

For distributed applications, sealed signed objects will be needed to guarantee both the authenticity and confidentiality of authorization tokens and other objects, especially SDOs, being transmitted across the network.

3.3.2 Guarded Objects

A *java.security.GuardedObject* is an object that encapsulates a protected object and a guard object. The guard object is responsible for restricting access to the protected object on the basis of the permissions held. When an attempt is made to extract the object being protected from the guarded object, an exception is thrown if the permission check performed by the guard object fails. Any object which provides the

method *checkGuard* method can act as a guard. All permission classes supplied as part of the JDK implement the *Guard* interface and so may act as guard objects.

At first glance, it might appear that a guarded object provides similar functionality to an SDO. In both cases there is some protected content and an associated guard / authorization object which performs access checking and so controls access to the protected content. However, a guarded object is used to determine whether the protected object may be extracted to allow its use. Once extracted, the object is no longer protected. An SDO, on the other hand, provides finer grained access control by determining which of its methods and data members may be accessed. Importantly, the access controls are located within an SDO so the protection is permanent.

When discussing guarded objects and in particular when describing an example illustrating how guarded objects could be used to protect a *FileInputStream*, Gong (Gong 1999) described the provision of SDO functionality as *"hypothetically we can radically rewrite the FileInputStream class as follows. ... For every access method (such as read(bytes)), the uniform security check in the form of g.getGuard() is invoked first."* Using this approach, the modified FileInputStream would have become a self defending object.

4 Applying the Self Defending Object Concept

When the design of an SAA is based on the SDO concept, the individual SDO objects that comprise the application are responsible for providing user centric access control on their resources. The majority of objects which comprise a SAA will not be self defending. Only those objects which encapsulate information or provide functionality which was identified within the application's security requirements, will be self defending. An important practical consequence of this observation is that the huge object libraries which are provided with a language compiler will not require modification to allow their use in an application employing the SDO concept.

A self defending object will normally use an authentication object to provide the required access control on its resources. However, if the access decision can be made on the basis of the possession of a privilege JAAS authorization could also be used. In either case, the authorization being checked will either be the right to invoke a protected method or accessing a protected data member in a particular manner. The authorization check will either succeed or fail. Within a method which accesses multiple data members, the situation might arise where access may be permitted for some data members but denied for others. Subject to the application's security policy, the SDO could respond in a number of ways including (1) rejecting the whole request (raise an exception) or (2) where only read access to the data members was requested, only the authorized subset of the data could be returned.

As the requested information is probably returned as an object, those data members to which access was not granted would be returned as dummy values (such as a null values or empty strings). Assuming that the 'main application' is error free, the substitution of the dummy data will not be apparent to the user, as that data will not be made available to the user. Should an error be present in the main application that would have permitted the unauthorized disclosure of that sensitive information, the action taken by the SDO has ensured that only the disclosure of dummy information can result. Such action by the SDO is appropriate, even if the returned object is a SDO, as 'defence in depth' is provided.

As indicated in Section 2.3, it is envisaged that RBAC, possibly supplemented by classic MAC, will be used by SDOs to control access to their resources. Consequently when a user is authenticated by the application, they will normally be required to authenticate both as an individual and as member of a role. The role information will be used for authorization while identity of the individual is primarily used for audit purposes. The actions of the individual, together with the role in which they are acting at the time of the attempted access, needs to be identified in the audit trail. Since the creation of the audit record is an integral part of the authorization process, SDOs are responsible for the generation of much of the applications audit trail. The determination of which actions should be logged by an SDO is of course be dictated by the security requirements of the application.

4.1 The Need for a Secure Environment

Although it is possible to develop a SAA which can provide fine grained access control that cannot be bypassed from within the application, such mechanisms cannot prevent access to the sensitive data via other means. Such application based mechanisms cannot prevent the copying or deleting of the files and/or databases which store the information being protected or the authentication and authorization information on which the protection mechanisms rely.[5]. The need for a secure environment was identified by Loscocco et al., (Loscocco & etal. 1998), and was expressed as *"Current security efforts suffer from the flawed assumption that adequate security can be provided in applications with the existing security mechanisms of mainstream operating systems"*. The authors continued by arguing that such an approach *"can only result in a "fortress built upon sand" "*.

Thus, the construction of SAAs, including ones which rely on the SDO concept, provide only a false sense of security unless the underlying operating system and hardware are secure. It is therefore necessary that any SAA reside in a trusted operating system environment, such as that provided by Trusted Solaris. The trusted platform is relied upon to secure the environment in which the application operates by providing such critical services as mandatory access control on the files. The fine grained access control provided by the application is complemented by and reliant upon, the services provided by its operating environment.

4.2 Making Authorization Decisions

As the protection is provided at the level of individual method calls and data member accesses, the huge number of authorization decisions being made requires that the authorization mechanism be lightweight. As one might expect, some form of credential will need to be presented to the SDO, to facilitate this mediation. Assuming that support from the JAAS framework is being employed, either a *Subject* object and/or a *AccessControlContext* object could be used as this token. In applications where the integrity of such tokens might be compromised, the cryptographic signing of those tokens should be considered. In those situations, the token passed to the

[5] An application potentially could ensure the confidentiality of such persistent data and be able to detect the modification of that data using cryptographic techniques. It would however be unable prevent its destruction nor would the quick recover from unauthorized modification be possible. Hence 'denial of service' attacks will remain a problem.

SDOs might be a *SignedObject* (Sect. 3.3.1) containing the 'actual authentication token'.

When an SDO method is presented with an authorization token, the SDO must determine whether that token entitles the principal to access the relevant resources of the SDO. Two approaches which are appropriate for making such an authorization decision involve the use of JAAS authorization or a custom authorization object. The remainder of this section discusses these choices.

4.2.1 Using JAAS Authorization

JAAS authorization which forms part of a standard Java™ distribution (e.g. JDK 1.4.2) provides a mechanism for making access control decisions purely on the basis of the possession of a privilege. As previously discussed (Section 3.2.3), by using an access control context object as the authorization token, the self defending object can determine whether the permission required to invoke that method or access individual data members in the specified manner, are held by the user.

4.2.2 Using a Custom Authorization Object

Access control decisions may be made by including authorization logic within the SDO and basing that decision on the identity of the user / principal making the request. That identity would be encapsulated within the authentication token, such as a *Subject* object, presented to the SDO. Such an approach has the advantage of allowing complex authorization logic to be used to make the decision rather than simply relying on the possession of a particular permission.

Rather than distribute that authorization logic throughout the SDO, it is desirable that the authorization and application logic should be separate by moving the authorization logic to a separate authorization object. That authorization object would encapsulate that portion of the application's access control policy which relates to its associated SDO. Logically, the authorization object need only provide two methods: one which determines whether the provided credential entitles the user to invoke the specified method of the SDO, while the other determines whether that user is entitled to access the specified data member in the specified manner.

The use of a separate authorization object has an additional benefit of simplifying the updating of the authorization logic in response to changes to the applications security policy. Of the numerous mechanisms that could be employed to implement an authentication object, probably the simplest of them will be discussed to illustrate the operation of an authorization object.

A single authorization class is written which uses an access control list to make its access decision. When instantiating its authentication object, an SDO will pass its fully-qualified class name as a parameter of the constructor. Thus the authentication object is loaded with the authorization information which pertains to that SDO from the application's policy file or database. Subsequent authorization decisions are based on that, now static, encapsulated policy information. Such a naive approach is only be suitable for short lived non-distributed applications as modifications to the security policy will not be reflected in existing authorization objects.

Other potential implementations include major extensions to the mechanism discussed above, whereby the updating of the security policy results in the use of that updated policy information by authentication objects or the direct implementation of the authorization logic within code of the authentication object (Barkley 1995).

4.3 Experience from Prototypes

Two small non-distributed Java™ applications were developed to determine the feasibility of using the SDO concept to develop security aware applications. Non-distributed applications were chosen at the prototype stage of the project to avoid the complications of distributed computing, including the secure migration of self defending objects and distribution of current policy information. These prototypes were developed using "Java™ 2 Platform, Standard Edition, v 1.4.1_2", which includes JAAS support within the distribution.

4.3.1 CD Store Prototypes

The reason for developing this prototype was primarily to act as a proof of concept i.e. determine whether the SDO concept could be successfully used as a design pattern for the development of security aware applications and evaluate its ease of use. User authentication relied on the user providing a username-role-password triplet. In this initial prototype, SDO authentication made use of a separate authentication object that used the naive approach described in Section 4.2.2. Both authentication and authorization information for this prototype were obtained from SQL databases.

It should be apparent that the access control policy used within this application was effectively one based on the possession of privileges. Consequently that application was later modified to explore different means of providing SDO based authorization utilizing JAAS support. Two additional versions of the application were developed: one of which used the security manager to perform privilege checking while the other preformed that task using an access control context. For compatibility with the default JAAS implementation, authorization information needed to be moved from the database to a suitably formatted policy file. Some minor modification also needed to be made to the applications authentication mechanism to make it compliant with JAAS requirements.

All variations of this prototype application maintain a small database of about 50 CDs which is meant to represent the titles held by a hypothetical retail store. The persistent CD data is held in a SQL database, with another copy being maintained in memory, the data model. This data model is implemented as a collection of CD objects. User requests for information are satisfied directly from the model, while modifications to data are reflected as updates to both the model and the underlaying database. As one would expect, the user interface permits the viewing, modification, insertion and deletion of CDs, the ability to search for CDs (by artist, ...) and the listing of the entire CD collection. Printing is supported from most screens. Images of the CD covers, which are displayed on the edit screen, are held as separate files.

The application was a SAA in that users acting in different roles are permitted to view and / or edit different information maintained for each of the CDs. All users were able to view at least some of the information held on all the CDs within the collection. Of the numerous classes used in the application only four classes were self defending: the data model, the class providing CD database access (the datastore), the file filter used to search for CD cover images and finally the CDs.

Significantly, in this application, no attempt has been made to tailor the user interface to the vari-

ous roles associated with the application. This means that throughout the GUI, an attempt is made to display all data held on a CD. Consequently, the application relies solely on the SDO to provide access control. This would correspond to a real application in which every application level access control measure was accidently omitted and would result in all sensitive information being disclosed. Despite this restriction, the objects providing the GUI are still responsible for the handling of exceptions raised as a result of authorization failures within the SDOs by the display of appropriate user messages.

Because the look of the user interface is independent of the roles associated the application, (fields do not appear or disappear depending on the role of the user) changes can be made to the applications security policy without the need to make any modifications to the application code. The other advantage of this approach for the prototype is that the completeness, or otherwise, of the SDO access control measures is immediately apparent.

An important requirement for an application is the ability to duplicate / copy an object. The situation is complicated if an SDO, rather than a normal object, is being copied. In an SDO, the user will often not have the right to access all of the data members of the object. The question arises, should the copy contain all data members, or just those that the user can access? It is our belief that to support the concept of 'defence in depth', the copy should contain only that information to which the user has access. When SDOs are copied, those data members for which read access was not permitted should be over-written with (known) dummy data. The copy method of CD objects was implemented in this manner. Consequently, whenever a CD object is retrieved from the model by the user, the retrieved CD object contains only the data which the user is permitted to access.

A similar approach was taken in the design of accessor methods of CD objects, they too return dummy data in the event of an authorization failure. It should be noted that all other authorization failures, in both CD and other SDOs, result in the raising of an exception.

As one might expect, special rights are required to initialize the application. A special 'creator' role needs to be created for this purpose, and a user acting in that role is needed to load the application. Associated with that role are significant rights which are not available in other roles such as the ability to instantiate the model and the datastore objects. Once the application loaded, that user is automatically logged out.

There were no difficulties experienced in using the SDO concept to implement the access control measures within the application[6]. The problem of ensuring that the access control measures were complete was not problematic as they were localized within the SDO. As expected, it was immediately obvious from viewing the code for an SDO, whether restrictions had been placed on the access to a particular method or data member. For these prototypes, the nature of those restrictions was determined by referring to the policy file / database.

4.3.2 Payroll Prototypes

The first prototype did not allow an investigation into inheritance between self defending classes. Consequently, a second prototype application was developed which implemented the payroll scenario described in Section 2.1.

The code structure for the application was essentially the same as that of the first prototype. On this occasion, SDO authorization relied on the use of an access control context to perform permission checking. This prototype supported the generation of an audit trail.

Inheritance within SDOs has some interesting consequences. In this application the only SDO inheritance hierarchy was: employee, manager and executive. To avoid the security problems associated by making data members protected, namely that the data members are available to any subclass and so vulnerable to attack(Gong 1999), all data members are private. This restriction has consequences when one self defending class inherits from another. Language semantics will necessitate the use of accessor/mutator methods to access the data members of super classes as they can not be accessed directly. This restriction has the side effect of ensuring that the access control restrictions enforced on a data member by its accessor and mutator methods, cannot be circumvented by a subclasses. A subclass can only enforce additional restrictions, not remove existing restrictions.

The incorporation of access control measures within self defending classes is not straight forward, when the classes inherit from one another. Consider the implementation of the getName method within the executive class where an employees' name is declared as a private data member of the employee class. At runtime, the getName method of an executive object, will invoke its counterpart in the manager object and indirectly in the employee object. If an authorization check is performed in each getName method, the call could be quite expensive if complicated authorization logic is required. The user must also possess the necessary rights to access the name of an executive, manager and employee. Provided the applications' security requirements are not contravened, this duplication can be avoided by only performing the check in the object in which the data member is declared. To illustrate the significance of this approach, consider the salary and name data members, both of which are declared in the employee class. Authorization checks need only be performed in the getName method of the employee object as the same security requirements apply to all three types of employees. However, as different security requirements apply to salary information, separate authorization checks are still required in the getSalary methods of the executive, manager and employee classes.

5 Future Work

To further investigate the usefulness of the SDO concept, additional research will be undertaken to explore the applicability of the SDO concept to larger scale systems, in particular those based on the use of software components, and later distributed systems.

Recent years has seen an increased adoption of the component technology approach to the construction of applications. Self defending components appear to be the natural extension of self defending objects and potentially provide assistance in the development of trusted software components (Meyer 2003). However for such an approach to be realized, a number of research questions relating to the behaviour of such self defending components need to be answered.

1. Is it possible to construct a reliable security aware application using third party components?

[6] As already discussed, because the prototype was GUI based, a very large performance penalty was incurred in that prototype in which the security manager was used to perform JAAS authorization. Additionally, its use resulted in the need to use an inelegant code structure.

2. How will a deployed SDC-based application system become aware of its operating environment and be able to make a reasoned risk assessment based on that environment? Based on that assessment, what action/s could an SDC-based application system take if the environment was found to be insecure?

3. How can a deployed component verify its own integrity?

4. What level of trust, (if any) can be assumed between the individual components of a SDC-based application system?

5. Should it be possible to permanently remove unwanted functionality from a SDC during composition to avoid this unwanted functionality from being exploited by external attackers or potentially hostile other components comprising the application?

Additional research needs to be conducted into how the effectiveness of the security measures can be quantified. Without such measures, it is impossible to provide a sound research based comparison of the effectiveness of the different ways of implementing an application's security measures, such as through the use the SDO concept presented in this paper.

6 Summary

The need for a change in the object oriented programming paradigm is necessary to provide enhanced application security required for SAAs. The concept of self defending objects was introduced as a means of simplifying the development of these SAAs and significantly improving the reliability of their security measures. A discussion of the support provided by the Java™ programming language for the realization of SDO concept was then presented and a discussion of how application developers might proceed using the SDO concept in their SAA development.

The usefulness of the self defending object concept in the development of security aware applications has been demonstrated through the development of small non-distributed applications which acted as a proof of concept. Further research exploring the applicability of the SDO concept to larger scale development based on component technology and the provision of trusted components, is being undertaken.

References

Barkley, J. (1995), Implementing role based access control using object technology, *in* C. E. Youman, R. S. Sandhu & E. J. Coyne, eds, 'Proceedings of the First ACM Workshop on Role-Based Access Control', Vol. 2, ACM Press, New York, NY, USA, pp. 93–98.

Gong, L. (1999), *Inside Java 2 Platform Security: Architecture, API Design and Implementation*, The Java Series, Addison Wesley, Reading, Massachusetts. The Java Series from Sun Microsystems.

Holford, J. W., Caelli, W. J. & Rhodes, A. W. (2003), The concept of self-defending objects in the development of security aware applications, *in* '4th Australian Information Warfare and IT Security Conference', Adelaide, Australia.

Loscocco, P. A. & etal. (1998), The inevitability of failure: The flawed assumption of security in modern computing environments, *in* 'Proceedings of the 21st National Information Systems Security Conference', pp. 303–314.

Meyer, B. (2003), The grand challenge of trusted components, *in* '25th International Conference on Software Engineering', IEEE Computer Press, Portland, Oregon, pp. 660–667.