An Aspect-Oriented Approach to Enhancing Multilevel Security with Usage Control: An Experience Report

Keshnee Padayachee and J.H.P. Eloff

Abstract— **The aim of this paper is to document experiences with augmenting multilevel security with usage control at the application level within the aspect-oriented paradigm. Multilevel access control is an access control policy that supports systems that process especially sensitive data. However, attribute-based access control is sometimes insufficient and needs to be combined with additional features in order to meet the demands of modern applications and systems. Usage control enables finer-grained control over the usage of digital objects than do traditional access control policies and models.**

***Index Terms*—Multilevel Security, Aspect-Orientation, Usage Control**

I. Introduction

Several authors have cited the benefits of using aspect-oriented programming (AOP) to implement security concerns (see [1] and [2]). As security is often extracted as a separable concern due to its orthogonal nature with respect to the functional requirements of a system, the separation-of-concerns principle of the aspect-oriented paradigm is suited to address security concerns [3]. Aspect-oriented software development is relevant for all major pillars of security – authentication, access control, integrity, non-repudiation – as well as for the supporting administration and monitoring disciplines required for effective security [4]. Multilevel security was once thought to be relevant only to military systems, but recently it has been gaining acceptance into other domains such as trusted operating systems, and in grid applications [5]. Maintaining the privacy of individuals is one of the most compelling reasons for implementing strong access controls in an organization. Research has shown that the efficacy of aspect-orientation in comparison to object-orientation can guarantee better security assurance when implementing multilevel security [5]. The aim of this paper is to explore the usability of aspect-orientation to implement multilevel security augmented with usage control.

Manuscript received 28/11/ 2006. Keshnee Padayachee is with the University of South Africa, School of Computing, Pretoria 0003 South Africa (e-mail: padayk@unisa.ac.za).
J.H.P. Eloff is with the Department of Computer Science, University of Pretoria, Pretoria 0002 South Africa (e-mail: eloff@cs.up.ac.za).

The fundamental benefit of using AOP within security is that it assists the abstraction of security-related programming tasks such as authentication, access control and integrity. These security concerns tend to crosscut objects. Crosscutting concerns are related issues that are scattered throughout the functionality of an application [6]. An additional benefit is that a security aspect may be reused for other applications [6]. For example, access control has similar requirements for most applications. Vanhaute and De Win [7] have demonstrated how to convert these security concerns into reusable generic aspects. Significantly, aspect-oriented software design is flexible enough to accommodate the implementation of additional security features after the functional system has been developed, as crosscutting concerns may be added or removed without making invasive modifications on original programs [8].

This paper investigates the strategy of using the aspect-oriented paradigm to facilitate the implementation of a non-intrusive, yet stricter enforcement of multilevel security in a functional system. The subsequent section discusses the concept of complementing multilevel security with usage control, while the discourse in the next two sections focuses on AOP and its influence on software security. Section 5 demonstrates and evaluates the aspect-oriented paradigm in terms of implementing multilevel access control in a fully functional system, while Section 6 concludes with directions for future work and insights gathered from the experiment conducted.

II. Complementing Multilevel Security with Usage Control

Multilevel Security was developed by the military and involves every object and user in the system being assigned a sensitivity label that consists of a level of secrecy and a set of compartments [9]. For example, the sensitivity label of a file is SECRET [ALPHA, VENUS], where SECRET indicates the level and [ALPHA, VENUS] the compartments. The security level is an element of a totally ordered set. The levels generally considered are TOPSECRET, SECRET, CONFIDENTIAL and UNCLASSIFIED, where TOPSECRET > SECRET > CONFIDENTIAL > UNCLASSIFIED [10]. The set of compartments is unordered. An access class c_i dominates ($\geq$) an access class c_j if and only if the security level of c_i is higher than or equal to c_j and the compartments of c_i include that of c_j.

Note that the term '*object*' does not imply 'object' in the typical object-oriented sense. In fact, the term 'subject' is the active process that requests access to the 'object', which refer to passive entities such as files or records. The Bell-LaPadula security model (BLP) is a formalization of the Multilevel Security policy that enforces two rules, namely the *No Read Up* and the *No Write Down* rules. These ensure that information flowing from a higher security level to a lower one is prevented. However, the BLP model displays a number of deficiencies [11], as is indicated below:

- 'Blind write-up' – The inability to inform low-security data whether a write to high-security data has happened correctly.
- 'Downgrading'– Moving information from a high-security level to a lower level is sometimes desirable.
- 'TCB bloat' – A large subset of the operating system may end up in the Trusted Computing Base (TCB).

Using AOP, it is possible to circumvent these problems, because the model for the control is embedded in the programming code and occurs while the application is executing. Problems such as 'blind write-up' and 'downgrading' may therefore be resolved within the code context when required. A semantic gap has been created between access controls of operating systems and programming languages because languages such as the Java Virtual Machine lack mechanisms to enforce such forms of mandatory access controls [12]. It is possible for AOP to fill this 'semantic gap' and thereby to reduce 'TCB bloat'.

Multilevel Security is a type of a mandatory access control scheme and was once thought to be relevant to the military only. However, multilevel access control is highly applicable in areas such as privacy, as 'access to privacy-sensitive data' can be regarded as analogous to accessing multilevel security data [13]. Even though users may be authorized to view information, this does not ensure that they respect the confidentially of the information that they have access to. Hence complementing traditional authorization with other forms of controls may influence individuals to behave in a trustworthy manner.

Sandhu and Park [14], recognizing the inadequacy of traditional access control models, proposed a new approach to access control called Usage Control (UCON). This model encompasses emerging applications such as trust management in a unified framework. They claim that the missing components of traditional access control are the concepts of *obligations* and *conditions*. *Obligations* require some action by the subject so as to gain or sustain access, e.g. by clicking the ACCEPT button on a licence agreement. *Conditions* represent system-oriented factors such as time-of-day, where subjects are allowed access only within a specific time period.

A family of models for usage control exists, involving pre-authorization and ongoing-authorizations. This paper will focus on the simplest model, the pre-authorization model, where a decision process is performed before access is allowed. Sandhu and Park [14] show how mandatory access control may be stated in terms of the UCON model, however only in terms of lattice-based authorization. In the Bell-LaPadula formalization of the multilevel security policy, security levels represent the pair consisting of the sensitivity level and the compartments. The security levels on objects are called *classifications* and the security levels on subjects are called *clearance*. The following represents a formalization of complementing multilevel security with conditions and obligations:

L is a lattice of security labels with dominance $\geq$
clearance: $S \rightarrow L$, classification $O \rightarrow L$
SUBJECT ATTRIBUTES = {clearance},
OBJECT ATTRIBUTES = {classification}

$$\text{allowed}(s,o,\text{read}) \Rightarrow \text{clearance}(s) \geq \text{classification}(o) \wedge \text{pre}\boldsymbol{B}(s,o,\text{read}) \rightarrow \{\text{true, false}\} \wedge \text{pre}\boldsymbol{C}(s,o,\text{read}) \rightarrow \{\text{true, false}\}$$

$$\text{allowed}(s,o,\text{write}) \Rightarrow \text{clearance}(s) \leq \text{classification}(o) \wedge \text{pre}\boldsymbol{B}(s,o,\text{write}) \rightarrow \{\text{true, false}\} \wedge \text{pre}\boldsymbol{C}(s,o,\text{write}) \rightarrow \{\text{true, false}\}$$

where *allowed(s,o,r)* predicate indicates that the subject is allowed to access object o with right r only if the indicated condition is true.
pre*B*(s,o,r) predicate determines if the subject has fulfilled obligations in order to access object o with right r.
pre*C*(s,o,r) predicate determines if the subject s is allowed to access object o under the current system conditions.

It is evident that this enforcement of security is very strong and possibly impractical, as it involves sensitivity levels, compartments, conditions and obligations. However this provides an opportunity to fully assess the flexibility of the aspect-oriented paradigm.

III. Background work on Aspect-Oriented Programming

An aspect is a modular unit of a crosscutting implementation that is provided in terms of pointcuts and advices, specifying what (advice) and when (pointcut) its code is going to be executed [15]. In terms of codification, aspects are similar to objects. However, aspects observe objects and react to their behavior [16]. An aspect is a piece of code that describes a recurring property of a program and can span multiple classes or interfaces [17]. Aspects improve the separation of concerns by making it possible to cleanly localize crosscutting design concerns. They also allow programmers to write, view and edit a crosscutting concern as a separate entity.

In the execution of a program, there will be certain well-defined points where calls to aspect code would be inserted [15]. These are known as join points. A pointcut is a set of join points described by a pointcut expression [18]. The pointcut is used to find a set of join points where an aspect code would be inserted. An advice declaration can be used to specify code that should run when the join points specified by the pointcut expression are reached [18]. The advice code will be executed when a join point is reached, either before or after the execution proceeds. For example, AspectJ supports *before*, *after* and *around* advices, depending on the time the code is executed [19]. A *before* in advice on a method execution defines code to be run before (after) the particular method is actually executed. An *around* advice defines code that is executed when the join point is reached and has control over whether the computation at the join point (i.e. an application method) is allowed to be executed or not [20].

Combining the application functional code and its specific aspects generates the final application. These two entities will be combined at compilation time by invoking a special tool called a weaver [17].

IV. An Aspect-Oriented Approach to extending Mandatory Access Control with Usage Control

De Win, Vanhaute and Decker [6] delineated aspects for discretionary access control within the aspect-oriented paradigm. Fortifying this access control model with multilevel security has been accomplished by Ramachandran, Pearce and Welch [5]. This paper considers how the usability of aspect-orientation could facilitate the process of complementing multilevel security (developed with aspect-orientation techniques) by means of usage control. It does not address the aspects of identification and authentication. Although Ramachandran, Pearce and Welch [5] developed an adequate aspect-oriented implementation of multilevel security, they do concede the following shortcoming: *'When a read or write to some stream type is intercepted, we have to access to the stream object in question. From this we must determine the true subject (i.e. the actual file being manipulated). Unfortunately, the Java API does not permit this directly (e.g. we cannot get back a file name from an instance of FileInputStream). To overcome this, we intercept creation points of these streams and manually associate with them the file name in question'*. This paper presents an alternative approach to circumvent this problem together with a proposal of extending aspect-oriented languages to resolve such problems. Furthermore, Ramachandran, Pearce and Welch [5] assume that the user's clearance is embedded within the original implementation. We assume that the original implementation contains no data relating security. We wanted to demonstrate that multilevel security policy may be totally separated from the original program, to assess the versatility, flexibility and extensibility of aspect-orientation. To this end, the aspect's constructor is involved in assigning security policies. The following statements define the `MLSAspect` aspect:

```
public aspect MLSAspect {
public MLSAspect(){
  //Assign security policies
}
pointcut Write(Object object, Subject
subject ): call(* Subject.write(Object) ) &&
args(object) && target(subject);
void around (Object object,Subject subject):
Write(object,subject){
if (cleareance(subject) <= classification
(object)){
    if (Obligations(subject,object,write)){
      if (Conditions(subject,object,write))
        proceed (subject,object);
    }
  }
pointcut Read (Object object, Subject
subject ): call(* Subject.read(Object) ) &&
args(object) && target(subject);
void around(String object, Subject subject):
Read (object,subject){
if (cleareance(subject) >= classification
(f)){
    if (Obligations(subject,object,read)){
      if ( Conditions(subject,object,read))
        proceed (object, subject);
    }
  }
boolean Obligations(){
    // Obtain Obligations}
boolean Conditions(){
    //Test Conditions}
}
```

The `MLSAspect` is a generalized aspect which implements multilevel security in terms of obligations and conditions. The aspect contains two pointcuts that represent each of the accesses, namely read and write accesses. Essentially these pointcuts will pick up all joinpoints in the program's execution that indicate a *Write* or *Read* access to a particular *Object* by a *Subject.* The aspect will allow the process to proceed only if all the authorizations, obligations and conditions are met. (Note: Object is not implied in the object-oriented sense and will probably indicate a file.) Significantly this aspect has *around* advices instead of *before* advices as specified in [5]. If *before* advices were used instead of *around* advices, then the aspect will allow the process to continue irrespective of whether the authorizations, obligations and conditions were fulfilled or not. Both *around* advices parallel the principles of *No read-up* and *No write-down.* This enforcement was subsequently complemented with the obligations and conditions requirements.

The next section will show a worked example demonstrating how the aspect may be extended for specific access control requirements.

V. A Worked Example

With respect to the extended multilevel security model described above, a small case study was generated to demonstrate the possibility that such a system may be fully implemented using AspectJ (ajdt_1.2_for_eclipse_3.0). The system initially contained only a single Personnel class to represent the Subjects of the system. Three access control requirements had to be enforced. Firstly, Personnel should be prevented from accessing files that they are not authorized to read from or write to as prescribed by multilevel access control policies. Secondly, whenever a Personnel member accesses a file, he/she should indicate whether he/she accepts the "Terms and Conditions" of accessing the file. Thirdly, no Personnel member is allowed access to files between 5pm and 6am. For example, Personnel Jane has a security clearance of SECRET [ALPHA] and the LOGISTICS file has a clearance of SECRET [VENUS, AL-

PHA]. According to the multilevel security policy, Jane should not be able to read file LOGISTICS. Accordingly, the `Read` *around* advice of the `MLSAspect` should not allow the following statement in application to execute: '`Jane.read("Logistics")`. However, Jane may write to `Logistics`, provided she accepts the obligations and abides by the timing conditions.

As a very simple system was built, it was easy to identify where access control needed to be applied. In short, one could surmise the joinpoints as the points where reading or writing operations were to be performed. The two pointcuts of the generalized `MLSAspect` were refined accordingly (see Appendix): `Write`, which intercepts all methods called '*write*', and `Read` which intercepts all methods called '*read*'.

It is essential for the objects of the system not to have any code relating to access control, as it defeats the purpose of using aspect-orientation. Therefore, the labeling of each object and subject also had to be confined to an aspect. This allows for ease of modification of sensitivity labels. The intertype declaration below provides a mechanism for tagging classes with a classification. This is a special aspect that allows additional data members and member functions to be included in a class without modifying the class itself. It saves on look-up time as opposed to inspecting an access control matrix instead. The following statements define the `ClassificationTag` aspect which assigns security levels to the `Personnel` class:

```
  public aspect ClassificationTag {
    private int Personnel.level;
    private Set Personnel.compartments = new
    HashSet();
    //relevant mutator and accessor
    //functions
}
```

In addition to tagging the user defined class `Personnel`, the file that is accessed by a `Personnel` object has to be tagged as well. Unfortunately tagging a Java class such as `java.io.File` cannot be implemented in the same manner as for the user-defined classes, since Java classes such as the `java.io.File` class or `java.lang.Object` class are "not exposed to the weaver:" The file classifications are controlled by a general aspect that associates the file name with a sensitivity label using a hash map. The following statements, defines an aspect `FileClassifictions` which assigns security levels to the file objects:

```
public aspect FileClassifications {
  private static HashMap hashmap = new
  HashMap();
    private static class FClassifications {
      private int level;
      private Set compartments;
      //appropriate accessors and mutators
    }
    public static void Add(String name, int
    level, Set s ){
      FClassifications FC = new
      FClassifications(level,s);
      hashmap.put(name, FC);
    }
    //appropriate accessors and mutators
}
```

This solution does not directly resolve the problem articulated by Ramachandran Pearce and Welch [5]. Instead, it is the object-oriented design that allows the aspect to access the file name. This illustrates an important lesson – although aspects may be designed after core functionally has been implemented, practical solutions are more probable if one iterates between the object-oriented design and the aspect-oriented design.

When designing security aspects, it is important to make them as generic as possible in order to facilitate reuse. In this particular case only read and write operations were considered, but this could have been expanded to other methods through the use of wildcards as proposed by Ramachandran Pearce and Welch [5].

The claim that aspect-oriented software design is flexible enough to accommodate the implementation of additional features after the functional system has been developed without making modifications on original program has been validated by the above experiment. However the challenges encountered does seem to indicate that implementing all nuances of the UCON model may not be feasible with AOP. It is questionable whether aspects could be used to implement ongoing-authorizations efficiently. It would be useful, if in addition to the *before*, *around* and *after* advice – AspectJ could offer a '*during* advice' which could allow for such 'parallel processing'.

We now attempt to address the problem identified by Ramachandran Pearce and Welch [5] by suggesting an extension to AspectJ. The limitations of AspectJ posed a problem in identifying the objects. It was easy to identify the classes used by referencing the variable called **thisJoinpoint** (a language construct of AspectJ) which contains reflective information about the current join point. It was, however, difficult to determine the name of the object itself, and it would have been ideal if the **thisJoinpoint** variable could be expanded to resolve this issue.

VI. Conclusion

The implementation of multilevel security within aspect-orientation allowed the access control features, together with the conditions and obligations, to be abstracted. If this implementation needs to be modified, it requires the consideration of only one separated modular unit, namely the `MLSAspect`. However, the cognitive understanding required to determine the interaction between intertype declarations and other aspects is challenging. More empirical studies need to be conducted to explore how implementation decisions made during the object-orientation design, either aid or impede the aspect-oriented design process.

APPENDIX

Listing A: Showing an example of MLS with Usage Control

```
public aspect MLSAspect {
public MLSAspect(){
  SetUp.SetFileClassifications();
}
pointcut Write (String file, Personnel p ):
  call(* Personnel.write(String) ) && args(file)
  && target(p);
void around (String file,Personnel p):
  Write (file,p){
  int pLevel = p.GetLevel();
  int fLevel = FileClassifications.GetLevel(file);
  Set pCompartments = p.GetSet();
  Set fCompartments = FileClassifica-
tions.GetSet(file);
  if
  ((pLevel <= fLevel)
  &&  (fCompartments.containsAll(pCompartments))){
    System.out.println(p.getName()+
      " allowed to write only to " +file);
      if (Obligations()) {
        if (Conditions())
          proceed(file,p);
      }
  }
  else{
    System.out.println(p.getName()+" cannot write to
"+file );
  }
}

pointcut Read(String file, Personnel p ):
  call(* Personnel.read(String) ) && args(file)
  && target(p);
void around(String file,Personnel p):
  Read (file,p){
    int pLevel = p.GetLevel();
    int fLevel = FileClassifications.GetLevel(file);
    Set pCompartments = p.GetSet();
    Set fCompartments = FileClassifica-
tions.GetSet(file);
    if ((pLevel >=  fLevel)
    && (pCompartments.containsAll(fCompartments))){
        System.out.println(p.getName()+
        " allowed to read only from "+file);
        if (Obligations()){
          if ( Conditions())
            proceed(file,p);
        }
    }
    else{
      System.out.println(p.getName()+" cannot read
from "+file);
    }
}
boolean Obligations(){
  InputStreamReader stdin = new InputStream-
Reader(System.in);
  BufferedReader console =  new Buffere-
dReader(stdin);
  String Answer = "";
  System.out.println("Do you accept the terms and
conditions required  to access this file ?");
  try{
    Answer = console.readLine();
  }
  catch(IOException ioex){
    System.out.println("Input error");
    System.exit(1);
  }
  if(Answer.equals("YES"))
    return true;
  else
    return false;
}
boolean Conditions(){
  Calendar cal = new GregorianCalendar();
  int hour24 = cal.get(Calendar.HOUR_OF_DAY);
  if ((hour24 >= 7) && (hour24 <= 17))
    return true;
    else
      return false;
}
}
```

REFERENCES

[1] B. De Win, F. Piessens and W. Joosen, "On the Importance of the Separation-of-Concerns Principle in Secure Software Engineering," in Workshop on the Application of Engineering Principles to System Security Design, 6-8 November 2002, pp.1-10.

[2] J. Viega and D. Evans, "Separation of Concerns for Security," in ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, June 2000, pp.126-129.

[3] P. Robinson, M. Rits and R. Kilian-Kehr, "An Aspect of Application Security Management," presented at AOSD'04 International Conference on Aspect-Oriented Software Development, Lancaster, UK, 2004.

[4] R. Bodkin, "Enterprise Security Aspects," presented at AOSD'04 International Conference on Aspect-Oriented Software Development, Lancaster, UK, 2004.

[5] R. Ramachandran, D.J. Pearce and I. Welch, "Aspectj for Multilevel Security," in The 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), 21 March 2006, pp.1-5.

[6] B. De Win, B. Vanhaute and B. Decker, "Security through Aspect-Oriented Programming," in Advances in Network and Distributed Systems Security, IFIP TC11 WG11.4 First Working Conference on Network Security, November 2001, pp.125-138.

[7] B. Vanhaute and B. De Win, "Aop, Security and Genericity," presented at 1st Belgian AOSD Workshop, Vrije Universiteit Brussel, Brussels, Belgium, 2001.

[8] N. Ubayashi, H. Masuhara and T. Tamai, "An Aop Implementation Framework for Extending Joint Point Models," presented at ECOOP' 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Oslo, Norway, 2004.

[9] E. Bell and L.J. LaPadula, "Secure Computer Systems: Unified Exposition and Multics Interpretation," Tech. Rep. Technical Report ESD-TR-75-306, 1976.

[10] D. Russell and G.T. Gangemi Computer Security Basics. O'Reilly and Associate, Sebastopol, California., 1991.

[11] R.J. Anderson. Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley, Computer Publishing, New York, 2001.

[12] V. Haldar, D. Chandra and M. Franz., "Practical, Dynamic Information Flow for Virtual Machines," in PLID'05 2nd International Workshop on Programming Language Interference and Dependence, 6 September 2005.

[13] W. Rjaibi and P. Bird, "A Multi-Purpose Implementation of Mandatory Access Control in Relational Database Management Systems," in Proceedings of 30th VLDB Conference, pp.1010-1020.

[14] R. Sandhu and J. Park, "Usage Control: A Vision for Next Generation Access Control," in The Second International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security, pp.17-31.

[15] F. Ortin and J.M. Cueva, "Dynamic Adaptation of Application Aspects," Journal of Systems and Software, vol. 71, pp.229-243, May 2004.

[16] J. Viega and J. Voas, "Can Aspect-Oriented Programming Lead to More Reliable Software," IEEE Software, vol. 17, pp.19-21, November 2000.

[17] J.P. Choi, "Aspect-Oriented Programming with Enterprise Javabeans," in Fourth International Enterprise Distributed Object Computing Conference (EDOC'00), September 2000, pp.252-261.

[18] D. Mahrenholz, O. Spinczyk and W. Schröder-Preikschat, "Program Instrumentation for Debugging and Monitoring with Aspectc++," in Proceedings of The 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, April - May 2002, pp.249-256.

[19] B. De Win, W. Joosen and F. Piessens. "Developing Secure Applications through Aspect-Oriented Programming." in Aspect-Oriented Software Development, Aksit, M., Clarke, S., Elrad, T. and Filman, R.E. Eds., Boston: Addison-Wesley, 2002, pp.633–650.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten and J. Palm, "Getting Started with Aspectj," Communications of the ACM, vol. 44, pp.59-65, October 2001.