AspectJ for Multilevel Security

Roshan Ramachandran
Computer Science
Victoria University of Wellington, NZ
ramachrosh@mcs.vuw.ac.nz

David J. Pearce
Computer Science
Victoria University of Wellington, NZ
djp@mcs.vuw.ac.nz

Ian Welch
Computer Science
Victoria University of Wellington, NZ
ian@mcs.vuw.ac.nz

ABSTRACT
A multilevel security (MLS) system has two primary goals: first, it is intended to prevent unauthorised personnel from accessing information at higher classification than their authorisation. Second, it is intended to prevent personnel from declassifying information. Using an object-oriented approach to implementing MLS results not only with the problem of code scattering and code tangling, but also results in weaker enforcement of security. This weaker enforcement of security could be due to the inherent design of the system or due to a programming error. This paper presents a case study examining the benefits of using an aspect-oriented programming language (namely AspectJ) for MLS. We observe that aspect-oriented programming offers some benefits in enforcing MLS.

1. INTRODUCTION
Multilevel security (MLS) [3] was developed by the US military in the 1970's to allow users to share some information with certain classes of user while preventing the flow of sensitive information to other classes of user. MLS achieves this by labelling data with classifications and assigning fixed clearances to users. The relationship between the classifications and clearances are used to determine if access by a user to data is permitted or disallowed. Note that the terms "clearance" and "classification" in the context of military systems refers to the security levels top secret, secret, confidential and unclassified. For example, a secret military plan will have different levels of information that correspond to various ranks.

MLS, once thought only to be significant to military systems, is also used in other domains like trusted operating systems, and in grid applications, where administrative users can set multilevel policies on their applications, thereby providing a fine grained control on the community users.

The Bell-LaPadula security model (BLP) [3] is a formalisation of MLS. BLP defines two rules which, if properly enforced, have been mathematically proven to prevent information at any given security level from flowing to a "lower" security level. These rules are called No Read Up (NRU) and No Write Down (NWD). The NRU rule states that a subject cannot read an object that has a higher security level. Whereas, NWD states that a subject cannot write to an object that has a lower security level.

To enforce MLS in a system, access control involves three entities: an object (the entity to which access is requested), a subject (the entity requesting access) and a reference monitor [1]. It is the responsibility of the reference monitor to check whether every access by the subject to an object is validated by the rules (i.e. BLP) set up by the MLS. To help in taking this decision, an object is assigned a classification and the subject is assigned a clearance.

Object-Orientation is an attractive approach to implementing MLS because objects are a natural way to represent system data and well-defined interfaces are a natural place to enforce access. However, there are several serious problems with this approach. Firstly, an object-oriented approach to implementing MLS can result in code tangling and scattering if access control is manually inserted into methods that read or write to sensitive objects. Although patterns such as the Proxy pattern can be used to structure object-oriented programs so that access control code is localised there are still maintenance problems because the relationship between the proxy and its object must be maintained at all times to prevent leakage of a pointer allowing uncontrolled access. More seriously, an object-oriented approach does not provide any support for preventing the introduction of security holes through poor design or bad programming. In other words, object-oriented programming does not support programmers in achieving stronger enforcement of security.

The aspect-oriented [10] approach is often perceived as an improvement over the conventional object-oriented approach in dealing with the issues of code tangling and crosscutting. Most previous work on implementing security using AOP have focused upon the concerns of reducing tangling and crosscutting [10, 12]. This paper describes how an AOP language (in particular AspectJ [2, 9, 10]) can actually go further to achieve stronger enforcement of MLS.

2. MOTIVATION
Figure 1 shows the class diagram of a payroll system with MLS. The payroll system is used by administrators and managers to track employee information, salary details, hourly rate etc. At the heart of the diagram is the PayrollSystem class, this implements functionality for adding new employees (Employee), changing their hours worked (WorkInfo) and changing their hourly rate (PayInfo). It is assumed that users of varying clearances access the payroll system and we want to control information flows between them. Note that we are not concerned with the problem of maintaining the integrity of the system — this would require the enforcement

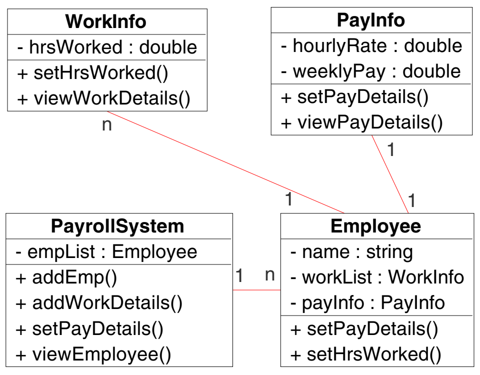

Figure 1: Class diagram of a payroll system

of a BIBA-style policy [4].

Consider a scenario where there are two types of users with different rights: managers and employees. Managers are allowed to view all information in the system, while employees can only view the non-sensitive information contained within the Employee class. Using BLP to enforce this requires setting up the system of clearances and classifications as follows: Employee instances are assigned a "low" classification, WorkInfo and PayInfo are assigned "high" classifications, managers are assigned "high" clearances and employees are assigned "low" clearances.

With this particular choice of classifications and clearances we can see that our access requirements described above are satisfied by enforcement of the BLP policy. The NRU rule restricts employees to reading Employee objects because all employees have clearances lower than the classifications of all objects except for Employee objects. While the NRU allows managers access to all objects because the manager's "high" clearance is equal or higher than all the classifications of all the other objects in the system. Furthermore, information cannot be passed by managers to employees because managers are prevented from writing to objects that can be read by employees (namely the Employee objects) by application of the NWD rule.

An object-oriented approach to implementing the payroll system would be to define a component that behaves as a reference monitor. This would take three arguments: the user's clearance, target object's classification and whether the requested action is a read or write operation. The reference monitor is responsible for deciding, based on the BLP rules, whether or not the user is authorised to access the object. For this to work, the programmer must either: insert access control code in all methods that read or write the protected objects (e.g. `setPayDetails()`); or create proxies containing the access checking code for the protected objects. Either way, he/she must *manually* identify those methods which need access control. This, of course, leaves open the possibility of programmer error — that some method may read/write a protected object without access control simply because the programmer missed it. Since humans are fallible, we believe it follows that an object-oriented approach is an inherently error prone way of enforcing MLS.

The problem is that we want code to be generated and added automatically based upon a declarative specification that defines what is a controlled information flow. Ideally this would be as finer-grained as possible to allow greatest control. Aspect-Oriented Programming offers such an approach and, we argue, can provide a stronger enforcement of security than is possible through conventional Object-Oriented Programming. This stems from the fact that AOP languages allow quantification over a set of *join points*. For example, we can capture all reads and writes to a given set of objects automatically using field `set` and `get` pointcut designators (as in AspectJ). Once we intercept a field read or write *joinpoint*, we can use *before advice* to perform the authorisation. This way, we avoid having to manually identify which accesses need authorisation. This guarantee provided by the quantification property of aspect-oriented languages is very difficult to achieve in traditional object-oriented languages.

```
abstract aspect BLPPolicy {

 abstract pointcut read(Object o);
 abstract pointcut write(Object o);

 // No Read Up (NRU)
 before(Object o) : read(o) {
  int oc = classification(o);
  int sc = clearance(Thread.current());
  if(sc < oc && sc != TRUSTED) {
   throw new SecurityException();
 }}

 // No Write Down (NWD)
 before(Object o) : write(o) {
  int oc = classification(o);
  int sc = clearance(Thead.current());
  if(oc < sc && sc != TRUSTED) {
   throw new SecurityException();
 }}

 abstract int clearance(Object o);
 abstract int classification(Object o);
}}
```

Figure 2: The generalised BLPPolicy aspect.

An AspectJ implementation of the BLP policy for the payroll system is shown in Figures 2 and 3. The two rules of the BLP model are encoded in an abstract aspect, where the concepts reading and writing are represented, but left unspecified. In this way, the policy can be (re)used simply by extending `BLPPolicy` and declaring these concepts appropriately (as done in Figure 3). In the implementation, we have simply used `Integer`s for the clearance and classification values; a more general approach might be to use an (abstract) lattice which the user would define appropriately. The use of the special `TRUSTED` status will be discussed later.

Our implementation shares some similarity with the AspectJ implementation of the Subject-Observer protocol [7]. One difference is that we have not made the roles (i.e. subject and object) explicit using interfaces as role markers (as done in [7]). This would have simplified our pointcut definitions to some extent. However, unlike application classes, AspectJ does not allow system classes to be modified, forcing us to use the somewhat less elegant approach.

3. CASE STUDY: FTP SERVER

This case study focuses on implementing MLS in a non-trivial (approx 20 classes), third-party application called jFTPd [8]. This im-

```
aspect PayrollPolicy extends BLPPolicy {

 pointcut read(Object o) :
   (get(* WorkInfo.*) || get(* PayInfo.*) ||
    get(* Employee.*)) && target(o);

 pointcut write(Object o) :
   (set(* WorkInfo.*) || set(* PayInfo.*) ||
    set(* Employee.*)) && target(o);

 int clearance(Object o) {
  // lookup thread clearance, possibly
  // resulting in user authentication
 }
 int classification(Object o) {
  // lookup object classification
}}
```

Figure 3: The payroll system's security policy.

plements an ftp server and we chose this as a case study for several reasons:

- An FTP server is a good example of an application requiring MLS enforcement. Users can read (download) and write (upload) files to the ftp server. Confidentiality needs to be taken care of to ensure secret files are only read by users with proper clearance. Likewise, users must be prevented from downgrading a file's classification (accidentally or otherwise).

- jFTPd is implemented in the object-oriented paradigm and has a built-in (albeit basic) security policy. This allows some comparison with the AOP implementation of MLS.

- jFTP is a third-party application with sufficient complexity that it might expose any limitations on the reusability of our abstract BLP policy aspect.

3.1 Overview of jFTPd

jFTPd is a Java implementation of an FTP server. jFTPd is multithreaded, so many users can connect and transfer files simultaneously. Figure 4 shows the original class diagram of jFTPd. For clarity, only classes concerning the access control for writing or reading files and basic ftp authentication are shown. The main class is *FTPHandler* which takes care of incoming connection setup requests. For every request, `FTPHandler` creates a new thread (i.e. `FTPConnection`) and assigns the incoming connection to it. This services all FTP commands from the client by delegating them to the `doCommand()` method (in `FTPConnection`). This in turn delegates onto a number of command-specific processing methods.

`FTPConnection` maintains a session for each connected user (i.e. `FTPUser`) and an object (i.e. `FTPSecuritySource`) which acts as a reference monitor containing the access control logic. Calls to this are scattered throughout the methods of `FTPConnection` to restrict certain types of access. Specifically, users may only access files in their home directory, whilst the anonymous user may be allowed (depending upon the configuration) certain access to files in the anonymous directory.

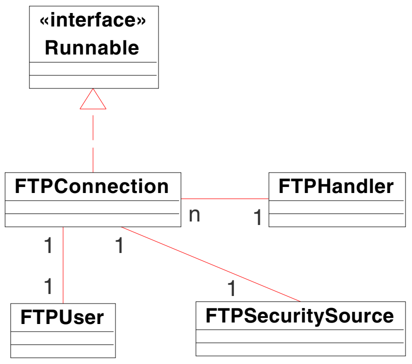

Figure 4: Original structure of jFTPd

3.2 AspectJ implementation

Our AspectJ implementation of the MLS security policy proceeds, as before, by extending `BLPPolicy`. The goal is to intercept all read and write operations to perform the required authorisation checks. But, what are the read and write operations in this case? In the previous example, the objects (object as in MLS) were `Employee`, `WorkInfo` and `PayInfo` instances, while the subjects were threads. In this case, however, the objects are files and the subjects instances of `FTPConnection`. Therefore, instead of using field `set` or `get` pointcuts to capture read or write operations, we must intercept all attempts to read or write files.

Figure 5 outlines the implementation. Several details have been omitted for brevity. The read and write pointcuts are defined in terms of the Java standard library calls for reading and writing streams. We must intercept on, for example, `InputStream` rather than `FileInputStream`; otherwise, accesses to an instance of the latter with a static type of the former will be missed. When a read or write to some stream type is intercepted, we have 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 APIs do not permit this directly (e.g. we get cannot get back a file name from an instance of `FileInputStream`). To overcome this, we intercept the creation points of these streams and manually associate with them the file name in question.

Threads other than instances of `FTPConnection` are regarded as having system/administration roles and given the special `TRUSTED` status (see Section 4.5 for more on trusted subjects). When a read or write operation is intercepted, we must determine the true subject (i.e. the user) in order to determine the appropriate clearance level. This information is maintained by `FTPConnection` as part of the original implementation. However, as outlined in Figure 4, instances of `FTPConnection` implement the `Runnable` interface (rather than extending `Thread`). This poses a problem as, although we can determine the thread responsible for a read/write operation via `Thread.currentThread()`, the Java API again provides no way to obtain the instance of `FTPConnection` it is associated with. As before, we overcome this by intercepting creation points for `FTPConnection` objects and maintaining their thread associations explicitly.

4. DISCUSSION

In this section we discuss our approach's granularity of control, how authentication is accommodated, the problems of covert channels, how improved modularity and maintainability are achieved,

```
public aspect JFTPdPolicy extends BLPPolicy {
 Map<Object,String> objects = ...;
 Map<Thread,FTPConnection> subjects = ...;

 pointcut read(Object o) : target(o) &&
  (call(* InputStream.read*(..)) ||
   call(* Reader.read*(..)) ||
   ...;

 pointcut write(Object o) : target(o) &&
  (call(* OutputStream.write*(..)) ||
   call(* Writer.write*(..)) ||
   ...;

 // *** OBJECT intercepts ***
 after(String s) returning(Object o): args(s)
   call(FileInputStream.new(String)) {
  objects.put(o,s);
 }

 after(File f) returning(Object o): args(f)
  : call(FileInputStream.new(File)) {
  objects.put(o,f.getName());
 }

 ... // other object creation intercepts

 // *** SUBJECT intercepts ***
 after(FTPConnection f) returning(Thread t)
  : call(Thread.new(Runnable)) && args(f) {
  subjects.put(t,f);
 }

 ... // other subject creation intercepts

 // *** BLPPolocy methods ***
 int clearance(Object o) {
  FTPConnection f = subjects.get(o);
  if(f == null) { return TRUSTED; }
  else {
   String user = f.getUser().getUsername();
   ... // lookup clearance for "user"
 }}

 int classification(Object o) {
  String file = objects.get(o);
  if(f == null) { return TRUSTED; }
  else {
   ... // lookup classification for "file"
 }}
 ...
}
```

Figure 5: An AspectJ-based MLS security policy for jFTPd.

the issue of trusted subjects and how to appropriately handle security failures.

4.1 Granularity of Control

In the case study we enforced security by intercepting system calls rather than application calls. For example, we intercepted file reads and writes rather than defining a read or write pointcut that intercepted application methods related to file access. The benefit of our system call approach is we get complete assurance that every read or write operation on a file has been validated by following the rules of No Read Up and No Write Down. Thus, we are not affected by the introduction of new application-level methods because these will use system calls to interact with the file system and therefore will be constrained by the BLP security policy. However, we are affected by changes to system classes that introduce new system calls. For instance, any newly introduced system API that deals with the reading or writing to files needs to be reflected in the existing pointcuts used to control access to system classes. However, we consider this to be less of an issue than the problems of traditional object-oriented languages, for several reasons: firstly, it's unlikely that system APIs will change frequently: secondly, there are fewer things which need to be identified by the programmer.

4.2 Authentication

In the case study we used existing jFTP authentication mechanisms to ensure that users are authenticated before making authorisation decisions. An open question is how to implement authentication within our framework. For example, should authentication be treated as a separate aspect or part of the authorisation aspect because it is a fundamental precondition for proper application of a security policy? The advantage of a separate aspect is that authentication could be easily plugged in and out, for example we could swap a local authentication mechanism for a federated authentication mechanism.

4.3 Covert Channels

Our current implementation does not address the problem of covert channels, that is information flows that are not controlled by a security mechanism. For example, a jFTP user with a “high” clearance could communicate with a user with a lower clearance by creating files with a “high” classification whose existence can be seen but the contents not read. This could be used to signal information read by the user with a higher clearance to the user with the lower clearance. To address this we could extend our implementation to hide the existence of files that cannot be accessed by the current user.

Another type of covert channel might arise if there are information flows not intercepted by our pointcuts. For example, in the case study we concentrated upon information flow via file reading and writing and assumed that no other information flows between threads was possible. However, what if we were wrong? Threads might have access to shared static fields and use these for communication. There are at least two solutions. We could use some form of static analysis to determine that flows between threads other than the ones we are dynamically checking do not occur. Unfortunately this might not be practical for all programs because of complexity. Alternatively, we could implement some form of dynamic pointcut for inter-thread information flow. This might be realized by intercepting all state changes (including variable reads/writes and parameter passing) and enforcing BLP. The disadvantage of this approach is the cost of intercepting all state changes. However, it may be possible to use it in conjunction with a more constrained form of static analysis to ensure enforcement is added only where threads may communicate with each other rather than throughout the application.

4.4 Modularity

By having the MLS access control logic in an aspect supports the well known AOSD notion of improved modularity and maintainability. This also makes it possible to have multiple authorisation aspects woven into the same base object. For example, in jFTPd, we could have a separate authorisation aspect executing after the `JFTPdPolicy` aspect to give finer control over anonymous users.

4.5 Trusted Subjects

There are sometimes legitimate reasons to relax the BLP policy for some subjects. Consider a listener thread that receives and dispatches requests to worker threads or a logging thread that updates a range of application objects. These threads may need to violate the BLP rules in order to function. Traditional MLS systems allow this by introducing the notion of trusted subjects who are allowed to violate the security rules.

In our case study this notion is satisfied because threads other than those associated with instances of `FTPConnection` are treated as trusted. We believe these threads are worthy of that trust simply because we have inspected their code to determine there are no violations of the BLP rules. This is not ideal, although it remains unclear how else such trust can be established.

4.6 Security Failures

Our current implementation raises a `SecurityException` when the BLP policy is violated. This may cause the client to crash if it does not expect this type of exception and it could also leave the application in an inconsistent state if the exception happens after a thread has updated one object but was denied access to another related object[1]. We can address the first problem by adding application-specific code to our aspects that create exceptions our client will be able to handle. The second problem is more problematic and may require the use of transactions to allow all-or-nothing semantics to be applied to controlled operations. The drawback of this is the requirement to understand the application semantics sufficiently in order to put all the security-related operations within a single transactional scope.

5. RELATED WORK

Implementing authentication and authorisation as crosscutting concerns for distributed object-oriented programs has already been addressed by several researchers, for example Lasagne [11], System K [12] and jFTPd [5, 6]. Lasagne implements an access control list style policy for a dating application. System K implements a MLS security policy for a third-party distributed editor. jFTPd implements an access control list style policy for a third-party FTP server.

These papers differ from our approach in that they use AOSD techniques with pointcuts that are tied to manually identified methods. This can lead to error if a method is missed or misidentified. Instead we concentrate on how to avoid this problem by determining pointcuts using fundamental properties of the application that are independent of functionality. For example, pointcuts tied to file update are better than pointcuts tied to specific methods believed to update files. This is an approach that was suggested by Welch and Stroud with respect to their implementation of MLS using a metaobject protocol[12].

[1]Consider a transfer of money between two different types of account.

6. CONCLUSION

In this paper, we have described an approach to incorporate MLS using aspects. In the case study, we used AspectJ to intercept Java library calls in order to provide a strong enforcement of MLS policy. By following this approach, we reduce the burden on programmers to correctly identify all the places in the code where authorisation is required. This quantification achieved by AspectJ is difficult to achieve in an object-oriented language.

We observe that even though the object-oriented paradigm provides a natural way to implement multilevel security, it falls short of preventing security flaws caused by bad programming or poor design. In this paper, we have demonstrated how aspects, in comparison to object-orientation, can guarantee better security assurance when implementing multilevel security.

7. REFERENCES

[1] J. P. Anderson. Computer security technology planning study. Technical report, ESD-TR-73-51, Oct. 1972.
[2] AspectJ home page. http://eclipse.org/aspectj/.
[3] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and multics interpretation. *MITRE technical report, MITRE Corporation, Bedford Massachusetts*, 2997:ref A023 588, 1976.
[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.
[5] B. De Win, W. Joosen, and F. Piessens. AOSD & security: A practical assessment. In *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.
[6] B. De Win, W. Joosen, and F. Piessens. Developing secure applications through aspect-oriented programming. In *Aspect-Oriented Software Development*, pages 633–650. Addison-Wesley, Boston, 2005.
[7] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *In Proc. conference on Object-Oriented Programming Systems, Languages and Applications*, pages 161–173. ACM Press, 2002.
[8] Ftp server with remote administration. http://homepages.wmich.edu/ p1bijjam/cs555
[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP 2001*, volume LNCS 2072, pages 327–353, Budapest, Hungary, 2001. Springer-Verlag.
[10] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.
[11] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. 23rd Int'l Conf. Software Engineering (ICSE'2001)*, 2001.
[12] I. S. Welch and R. J. Stroud. Re-engineering security as a crosscutting concern. *Comput. J*, 46(5):578–589, 2003.