Security Policy Enforcement in the OSGi Framework Using Aspect-Oriented Programming

Phu H. Phung David Sands
Chalmers University of Technology, Sweden*

Abstract

The lifecycle mismatch between vehicles and their IT system poses a problem for the automotive industry. Such systems need to be open and extensible to provide customised functionalities and services. What is less clear is how to achieve this with quality and security guarantees.

Recent studies in language-based security – the use of programming language technology to enforce application specific security policies – show that security policy enforcement mechanisms such as inlined reference monitors provide a potential solution for security in extensible systems. In this paper we study the implementation of security policy enforcement using aspect-oriented programming for the OSGi (Open Services Gateway initiative) framework. We identify classes of reference monitor-style policies that can be defined and enforced using AspectJ, a well-known aspect-oriented programming language. We demonstrate the use of security states to describe history-based policies. We also introduce and implement various levels of security states in Java to describe session level history versus global application level history. We illustrate the effectiveness of the implementation by deploying the security policy enforcement solution in an example scenario of software downloading in a standard vehicle system.

1. Introduction

Vehicle telematics and infotainment systems have traditionally provided fixed functionality. The problem with this state of affairs is the lifecycle mismatch between the vehicle and its software. IT services need to be dynamic; what is appropriate today can be completely outdated in 6 months let alone 6 years. The current goal is to enable truly open systems which make it easy to add third-party services.

However, such extensible systems face difficult security problems. To get the full benefits of extensibility in such systems one needs to allow potentially untrusted applications access to security sensitive resources. A simple sandboxing view which grants all-or-nothing access to a static set of resources, determined on the basis of trust, is too course grained. To be flexible we need to be able to enforce application-specific policies.

For example, suppose a client wishes to install a third-party service to an on-board vehicle computer, and the service needs to be able to send SMS (text) messages in order to function properly.

There are possible problems: the program could be malicious and deliberately send too many messages e.g. to a high-cost service. Or the application may simply have bugs, causing it, under certain circumstances, to repeatedly send messages. In the standard security approach one must decide, on the basis of trust (e.g. established via digital signatures) whether to permit the application access to the SMS service. This all-or-nothing approach has obvious limitations. In an open environment it is hard to establish meaningful trust relationships, and even when one can, trust is not equated with quality.

But access control is probably not the real security policy that we are interested in – it is really just an implementation mechanism. A more fine-grained security policy for such an application – one which can be stated independently of any trust relationships and access control mechanism – might be something like the following: allow a third party application to access the SMS service, but:

- restricted to a specific recipient address,
- with a limit on the number of messages sent per day, and
- depending on the vehicle's location.

To accommodate the last point using a standard access control mechanism one might have to additionally permit an application access to GPS location data, thus opening up a host of additional privacy related vulnerabilities. Another alternative to enforce such a fine grained policy would be to push the security into the API itself. A limitation with that

*Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. {phung,dave}@cs.chalmers.se

0730-3157/08 $25.00 © 2008 IEEE
DOI 10.1109/COMPSAC.2008.149

approach is the need to predict the way in which we might wish to control the API at its inception.

One general approach to imposing such a security policy on an otherwise untrusted system is to use a *reference monitor*. The concept of a security reference monitor [9] is a classic method to specify and implement secure systems. The reference monitor is a trusted component which intercepts security relevant resource requests and applies a security policy to decide whether to grant such requests. There has been considerable interest recently in using reference monitors at a purely software level – for example by rewriting software to "embed" (inline) a security policy within it – to provide expressive and efficient application specific security policies for software components e.g. [15, 22].

In this paper we consider the application of this inlined reference monitor approach in the context vehicle telematics/infotainment systems – on-board vehicle computer and communications systems. The study considers a new combination of methods. We consider the OSGi (Open Services Gateway initiative) standard [4] as a representative open middleware platform for telematics systems. The main questions addressed in this case study concern the architecture and implementation of reference monitors for third-party applications. To implement reference monitors we adopt a language-based approach using *aspect-oriented programming* with the AspectJ compiler [1, 20], rather than a more security-specific program rewriting tools such as e.g. PoET/PSLang [15] or Polymer [22]. A strength of this approach is that it uses a relatively complete and well-tested tool. A weakness, in principle, is that is does not provide direct support for policies (aspects) which make reference to the computation history. The main questions considered in this study are

1. What classes of reference monitor-style policies can be enforced using AspectJ?
2. How can this approach be integrated with the OSGi platform without making platform modifications?
3. What are the shortcomings of using AspectJ for implementing reference monitors?

Organisation The next section briefly reviews the background material for this study, including the reference monitors approach, aspect oriented programming and the AspectJ language, and the OSGi framework. Related work is also reviewed in each field. In Section 3, we categorize classes of security policies, and discuss the issues of various levels of security states, history-based policies (Question 1 above). In Section 4 considers aspects of deployment the security policy enforcement in the OSGi framework using AspectJ, and deployment architecture (Question 2). In Section 5 we conclude our contributions, and discuss limitations (Question 3) as well as further research issues.

2. Background and Related Work

There are three main strands of background material for this study, and these are briefly reviewed in this section. We also review the related work. Firstly we discuss how reference monitors can be implemented by transforming programs so that security checks are inlined in the code – the so-called *inlined reference monitor* approach. Secondly we review the specific technology we adopt here: aspect oriented programming, and, more specifically, the AspectJ language. Thirdly, we describe the middleware framework in which we apply this study: the OSGi framework.

2.1. Security Policy Enforcement by Program Transformation

Security policy enforcement by program transformation is an implementation of the reference monitor approach in which a target program is modified so that it will adhere to a security policy when it executes. More specifically, new code will be added in security-relevant actions or events to check the program respects the security policies. Thus, the modified program is guaranteed not to violate the policy. The mechanism needs a language to describe the security policy and a re-write tool to modify a target program. A number of proposals have been directed at specifying more expressive classes of security policies and implementing the appropriate tool to rewrite a target program so that the policies are guaranteed to be enforced in the rewritten program. PSLang/PoET [15] is such a language/tool for security policy enforcement in Java bytecode. The implementation of PSLang/PoET is based on security automata [25] software fault isolation: if the original program is about to violate the security policies, the modified program will halt instead. Polymer [22] allows more expressive security policies for Java applications. Polymer provides more powerful transformational responses to application events. In general such a mechanism is useful for protecting extensible systems since it does not rely on a complex tool chain (c.f. the proof carrying code approach [26]). However, one current disadvantage of these tools is that they are research prototypes and lack the robustness and completeness (e.g. in terms of source language features) of mature industrial tools. This puts limitations on the kinds of experimental investigations that one can currently perform in a real context like the OSGi framework.

Our approach is to implement policy enforcement using AspectJ, an "industrial strength" aspect oriented programming language. This has benefits of providing a complete and robust tool which can be applied at an appropriate level for this study (i.e. Java bytecode). This choice also presents some challenges and problems that will be discussed in this article. The next subsection gives overview of aspect-

oriented programming and the AspectJ language.

2.2. AOP and the AspectJ Language

Aspect-oriented programming (AOP) [20] is a new programming paradigm providing programmatic means to modularise cross-cutting functionalities of complex software systems so that program concerns in a software system can be captured and encapsulated in to so-called aspects. AspectJ is a language that extends Java and implements the paradigm of AOP. In AspectJ, an *aspect* comprises a *pointcut*, which defines the point and the condition under which the aspect modifies the behaviour of an application, and an *advice*, which defines what modifications should be applied. Pointcuts are sets of execution points in Java such as object constructors, method calls, method executions, variable settings, and so on. Advice is expressed in the traditional Java language. In order to apply aspects in a target Java program, an aspect developer defines aspects, then uses an aspect tool to combine the target program and the aspects. This stage is called *weaving*, where the target program will be analysed and modified by matching pointcuts and inserting advice.

These features of AspectJ make the language and the weaver tool suitable as a security policy enforcement tool. In general, security policies could be defined by aspects with security-relevant events declared in pointcuts and security responses defined in advice. However, whether an aspect language like AspectJ can scale up to real systems as a policy language is still the challenge. For instance, what sorts of security policies can be described in AspectJ? How mature is the security assurance provided by AspectJ? What are the shortcomings of AspectJ in the sense of security policy enforcement that needs to be investigated in further studies? In related work, several proposals for building secure software systems using aspect-oriented design have been surveyed in [12]. Basically, these proposals only introduce the aspect-oriented approach as part of the general design and development of security requirements for a software system. This work addresses the questions by studying the use of AspectJ in the context of dynamic security policy enforcement for the OSGi framework in telematics systems.

2.3. The OSGi Framework in Vehicle Systems and its Security Challenges

The OSGi (Open Services Gateway initiative) [4] is a framework implementing a complete and dynamic component-model that is missing in stand-alone Java virtual machine (JVM) environment. An application in OSGi consists of one or more components, called *bundles*. Bundles can be installed remotely and can be started, stopped, updated and uninstalled without restarting the JVM. The OSGi framework offers a co-operative model so that bundles can discover and use services provided by others in the same OSGi framework. The OSGi framework has been used in in-vehicle systems by several car manufactures. For example, BMW used the OSGi specifications as the base technology for its high-end infotainment platform [8]; the GST project defines an application runtime environment for a client system (vehicle) using the OSGi framework (c.f. [6], Open Systems Implementation Guide).

The OSGi framework sits on top of a JVM and its security mechanism is based on the Java 2 security model [5]. The main addition is simply the ability to authenticate bundles to be able to verify bundle integrity. In other respects it has the standard advantages of the Java security model (e.g. memory safety), but also the standard limitations as discussed in e.g. [14, 16]. In particular, the access control model – as discussed in the introduction – restricts the functionality of mobile code since they adopt an "all or nothing" access to computing resources. Code signing mechanisms only certify the origin and the integrity of code. Regarding the expressiveness of Java's security mechanism, policies depending on the history of executions (other than that visible by stack inspection) or value of variables at runtime cannot be defined in Java 2.

In the context of security for the OSGi framework, some research [23, 24] has investigated secure bundle deployment. The solutions help certify the origin and the integrity of code. More relevant to the present work is [17] which describes a rule-based runtime monitor integrated into the OSGi to detect and prevent certain security violations. This proposal, of course, has a potentially high runtime cost. To the best of our knowledge, no prior studies has investigated on security policy enforcement for OSGi using aspect-oriented programming as we consider in this article.

3. Classes of Security Policies in AspectJ

In this section we categorize some of the classes of security policies that we can encode in AspectJ. We then discuss other issues such as history-dependent policies and levels of security states. Dealing with multiple threads and interacting among security policies are also discussed.

3.1. Security Policies by Response Actions

In AspectJ, an aspect contains a pointcut and an advice. Mapping from AspectJ to a security policy language, a *pointcut* is a definition of an application's *security-relevant event*, and an advice is a *security decision* at a defined event. A variable in aspect objects could be considered as part of what we will call the *security state* to keep track of the history of execution and application's activity. Since runtime parameters can be accessed at a pointcut, security policies in aspects could be specified dynamically.

The question that arises at this point is: what kinds of security policies can be described. This subsection categorises kinds of security policies based on kinds of response actions (*security decision*) to security-relevant events. The categorisation is inspired by the edit automata [21], a theoretical work which classifies rich enforceable security policies. The edit automata view goes beyond the classical reference monitor approach because it proposes powerful transformational abilities such as the ability to *suppress* actions, *replace* actions, *insert* new actions, and to *truncate* execution.

In each of the following examples we illustrate one of these policy types to show that they can be represented directly in AspectJ. The examples are written in pure AspectJ (and Java) code; the pure AspectJ code should be self-explanatory.

1. *Suppression:* This represents prohibiting an action by simply suppressing (ignoring) it. This is suitable for actions whose completion is not critical for the functionality of the system. This kind of policy could be defined by the advice **around**. For example, the policy *"suppress the alert message when the vehicle speed is over 80mph"* could be defined in AspectJ as:

Listing 1: A suppression policy example.

```
1  import osgi.VehicleSystem;
2  public aspect Suppression{
3    pointcut alertMessage():
4          (call (* VehicleSystem.alert(..)));
5    void around():alertMessage(){
6       if(VehicleSystem.speed()>80){
7          VehicleSystem.log("alert message " +
8                            "during high speed");
9       }
10   }
11 }
```

2. *Insertion:* Action is allowed but requires insertion of additional code before or after execution.
   To insert a sequence of actions in a target program, e.g. logging information, both the advice **before** and **after** could be used depending when we want to insert the actions. The listing 2 gives an example that stores the bundle object to the instance of the class library BundleHandler before a bundle starts in the OSGi platform.

Listing 2: An insertion policy example.

```
1  import osgi.*;
2  import org.osgi.framework.BundleContext;
3  import org.osgi.framework.BundleActivator;
4  import advice.BundleHandler;
5  public aspect BundleStart {
6    BundleHandler bundle;
7    pointcut startBundle(BundleContext context):
8    execution(* BundleActivator+.start(BundleContext))
9                                    && args(context);
10   before(BundleContext context):startBundle(context){
11     bundle = new BundleHandler(context);
12     VehicleUtils.log("A bundle starts");
13   }
14 }
```

3. *Truncation:* This corresponds to the classic runtime monitor approach: if the application attempts to perform a prohibited action then execution will be aborted.
   This kind of policy could be defined by the advice **before** or **after**. In the body of the advice, security policy developers could check whether the policy condition matches, then the current application could be stopped by calling a method of a library. Listing 3 in AspectJ (only shows the advice) illustrates the policy *"stop the application if it attempts to operate the brake system"*[1].

Listing 3: A truncation policy example.

```
1   before():call (* VehicleBrake.brake(..)){
2     VehicleUtils.log("The application attempts"+
3       "to operate operates the brake system");
4     try{
5         bundle.stop();
6     }catch(BundleException e){}
7   }
```

   The object bundle in Listing 3 is an instance of the class library BundleHandler and it is assumed to be initialized previously, e.g. in the listing 2. Class BundleHandler is an auxiliary library class for supporting security advice. The class is system-dependent, e.g. in this example the class is implemented for the OSGi framework. The method stop() will be responsible for stopping the corresponding bundle.

4. *Replacement:* An action should be replaced by a safe alternative action.
   The advice **around** could be used to define this kind of policy. This advice is similar to the suppression advice but instead of logging an error, this advice should return a new action (implemented somewhere in the aspect file or in a library) having the sufficiently similar functionality as the action we want to replace (e.g. to prevent the application from crashing). The following example would *"replace the method call send(..) by the new method secureSend()"*;

Listing 4: A replacement policy example.

```
1  public aspect Replacement{
2     int around():(call (* send(..))){
3        return securedSend();
4     }
5  }
```

3.2. Dealing with History-Dependent Policies

Some response actions of program events in a security policy may depend on the history of the program execu-

[1]The example is rather tongue-in-cheek: we are not suggesting that systems will be *that* open.

tion. The Chinese Wall policy [10] is such a policy example where information access control is decided on the basis of the earlier access. AspectJ supports a mechanism to capture the current call stack but cannot capture the history of the earlier program execution. To deal with problems of history-dependent aspects, recent work, for example [7, 11, 18], has focused on defining aspects that support to directly observe the history of a computation. While this would certainly be useful in the present context, we have chosen to define security policies in "standard" AspectJ. The method to do so is straightforward. We use variables as security states to record appropriate parts of the history of the program execution in order to define history-based security policies. Response actions can use the states in their security decisions. The policy example in Listing 5 demonstrates the use of pure AspectJ in describing history-dependent security policies.

3.3. System Level and Application Level Security States

In a security automaton, decisions of response actions are based on security states at runtime. As we mentioned above, such states could be encoded by local variables in aspects. However, some policies require data both from the global *system* level as well as the *application* level. For instance, a global policy (across all applications) that allows each *application* to send 3 SMS messages per day, but limiting the total number of messages of the whole *system* to 10 per day requires both the data of the number of the sent SMS messages in each application and in the system. AspectJ only support session level states (local variables per run), therefore, we have implemented a library to encode system level states (**class** GlobalState) and application level states (**class** ApplicationState). Each state level is encoded in a file, and each file is monitored by appropriate daemon thread bundle. The daemon thread implements the temporal component of the policy to reset the state, i.e. the SMS count, when the period has elapsed. The value of each state is updated and synchronized via the files by the library classes and daemons. Using the classes, policy writers could define different levels of states in security policies. The following AspectJ code illustrates the above policy example using the classes.

Listing 5: A policy example illustrates the use of different levels of security states.

```
1  import advice.GlobalState;
2  import advice.ApplicationState;
3  import advice.Duration;
4  public aspect StateExample{
5   pointcut appStart():execution(* start(..));
6   ApplicationState astate =
7                          new ApplicationState("SMSApp");
8   before() : appStart(){
9     if(!astate.existState("SMSNum")){
10      astate.createState("SMSNum");
11      astate.setStateDuration("SMSNum", Duration.Day);
12      astate.setStateValue("SMSNum",0);
13    }
14  }
15  pointcut SMSsend(): execution(* SMS(..));
16  private void SMSIns(){
17    Integer appSMS =
18      (Integer)astate.getStateValue("SMSNum");
19    Integer sysSMS =(Integer)GlobalState
20                                  .getStateValue("SMSNum");
21    astate.setStateValue("SMSNum",appSMS.intValue()++);
22    GlobalState.setStateValue("SMSNum",
23                                       sysSMS.intValue()++);
24  }
25  int around():SMSsend(){
26    int appSMS= ((Integer)
27      astate.getStateValue("SMSNum")).intValue();
28    int sysSMS=((Integer) GlobalState
29                    .getStateValue("SMSNum")).intValue();
30    if((appSMS<3)||(sysSMS<10)){
31       SMSIns();
32       return proceed();
33    }else{
34      System.err.println("Policy violated");
35    }
36  }
37 }
```

In Listing 5 example, we use one application-level and one system-level state variable, both labelled with "SMSNum". The application state object is initialized at the beginning, line 6-7, and the "SMSNum" state is created if necessary in line 9-13. Notice that global states must be initialized and configured when the system starts. We assume that the global state has been initialized elsewhere. The policy is checked in line 30 inside the advice **around** (line 25); if the policy is violated, the SMS function will be suppressed (line 33-35), otherwise states are updated (line 31, call the method in line 16-24), and the function is executed (line 32).

Dealing with multiple threads The security states are encoded and synchronized via files, so multiple threads (each application in the OSGi is a thread running on the framework) should access common states under mutual exclusion.

Interacting among security policies Different security policies for an application could use the same application-level state, thus the policies can interact with each other by reading and writing states. System-level security states allow security policies of different applications to interact. These issues, which to our knowledge have not been addressed in prior security policy enforcement proposals, are important for multiple applications running on the same framework like the OSGi.

4. The Deployment of Security Policy Enforcement in OSGi using AspectJ: A Case Study

In this section we describe the system architecture (4.1) – in broad terms – for weaving security policy into third-

party bundles to be deployed in the OSGi framework. We discuss some deployment issues (4.2) of security policy enforcement in the OSGi using AspectJ.

4.1. System Architecture and the Scenario

The Global System for Telematics (GST) project [6] provides a reference standard for vehicle systems. The standard is J2ME/OSGi based, describes how a telematics client application can be downloaded and installed over the air from a control center, and specifies an interface for receiving vehicle data. In this study, we use the architecture described in the standard, and the Knopflerfish open source OSGi framework [3] for the in-vehicle system.

The first questions are how and when the security policy, as represented by aspects, will be woven into the program. One possibility is to extend the framework itself to allow bundles to be recompiled using the AspectJ compiler when they are downloaded. This approach has the drawback of requiring potentially a lot of modification to the OSGi client, but also a significant computational load (compilation) to a potentially small computing device. Our approach is to proxy the downloading of an application via a trusted control center who takes case of the weaving on behalf of the client.

Consider the following example practical scenario. A hotel service company offers an infotainment application for in-vehicle systems that provides useful information about hotels near by the vehicle location. A driver can use this application in his truck to find a suitable hotel when spending a night on the way of his transportation. To install the application, a driver makes a corresponding request to the control center as in the GST standard. One new stage in our scenario is that before installing over the air to the in-vehicle system, the control center weaves the application with defined aspects (considered as security policies) to make sure that the application does not violate the desired security policies. Thus, the application has been modified following aspect-oriented mechanism and it is guaranteed that the security policies described in aspects are enforced thanks to the weaving process in the control center. Figure 1 illustrates steps of the scenario.

What is left open here is the choice of policy; one possibility is that the policy is sent from the client together with the bundle request. There are also possibilities for including some centralised policy control – e.g. mandatory "minimum" policies enforced by the control center.

4.2. Deployment Remarks

In the deployment scenario, the control center weaves application bundles with defined aspects (as security policies) by using the AspectJ weaver tool [1]. One additional

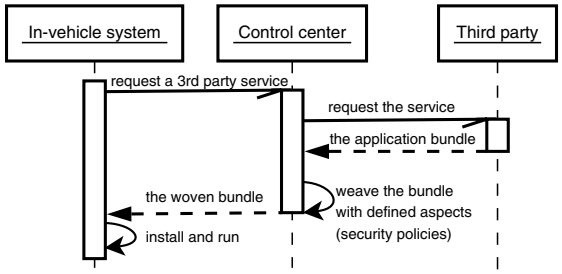

Figure 1: The weaving process scenario

stage in this process is to modify a bundle's manifest file so that the bundle can use AspectJ library packages and the additional library packages at runtime. In the in-vehicle system, the woven bundles need the AspectJ library and our library packages, so the libraries must be installed in the framework before installing and running a woven bundle. We package our library in a jar file with a manifest file that allow bundles access the library, then install the library to the framework. The AspectJ library is also installed after re-packaging with a modified manifest file that allow bundles access the library packages.

Test example We implemented a simple application bundle simulating the hotel guide service described above. The bundle communicates with a simple server application to get hotel information, and to make booking. Simple security policies reflecting various identified classes of policies described in AspectJ (policies could be defined in one or separate aspect files) are used to weave the bundle so that the bundle is modified to enforce the defined policies. The woven bundle was re-deployed and run successfully on the Knopflerfish OSGi framework. Several test cases were performed to illustrate that the defined security polices are correctly enforced for the bundle.

5. Conclusion and Further Work

Conclusion This paper has demonstrated the use of an aspect-oriented language, AspectJ, in the context of security policy enforcement. We have illustrated how various sorts of security policies are categorised and described in AspectJ as advice. Our demonstration has resulted in the first study of security policy enforcement using an aspect-oriented language in an open system like the OSGi framework. The study differs from research of security policy enforcement in that it is based on the more industrially well-known language AspectJ and the main stream Java language without defining any new policy languages. We also design and implement a library can be used to describe different levels of security states such that history-dependent security policies

could be defined, even in the presence of multiple interacting threads. Security policy definition and enforcement by aspects and libraries were deployed and tested in an open source OSGi framework. As a result, we could conclude that the security assurance provided by the policy enforcement mechanism using AspectJ is promising (and certainly adequate for small examples) and can be deployed in the OSGi framework.

Further Work In the small-scale examples that we investigated we did not encounter problems with representing history information explicitly. Whether this becomes a problem for larger examples remains to be seen. Another feature that we did not investigate is the composition of different security policies. Several ongoing research works on composition of aspects are expected to be directly applicable in this context [13]. Future work should also consider the design of policy management mechanisms, dealing with issues such as interactive policy construction.

At the level of the framework, we assume that the weaving process (security policy enforcement) in the control center is done automatically by running a script. Alternatively, the weaving process could be executed in the in-vehicle system side automatically by integrating the AspectJ weaver tool in to the OSGi framework. Recently, Equinox Incubator - Aspects project [2] provides an Eclipse plug-in combining the OSGi runtime of Eclipse together with AspectJ. A more recent work [19] integrates AOP into OSGi by enabling customised load-time weaving with OSGi. However, these are intended to support general AOP development in OSGi, not to focus on security policy enforcement. In future work, we will investigate on this integration to support "online" security policy enforcement for the OSGi framework at in-vehicle systems.

Acknowledgements Thanks to Marcus Larsson at Volvo Technology (VTEC) and the anonymous referees for helpful comments. This work was partially funded by Vinnova (Swedish Governmental Agency for Innovation Systems), project SESAME.

References

[1] The AspectJ Project. `http://www.eclipse.org/aspectj/`.

[2] Equinox Incubator - Aspects. `http://www.eclipse.org/equinox/incubator/aspects/`.

[3] Knopflerfish - Open source OSGi. `http://www.knopflerfish.org/`.

[4] OSGi Alliance, OSGi - The Dynamic Module System for Java. `http://www.osgi.org`.

[5] Sun Microsystems, Java Security Architecture. `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-specTOC.fm.html`.

[6] The Global System for Telematics (GST) project. `http://www.gstforum.org`.

[7] C. Allan et al. Adding trace matching to AspectJ. Technical report, The abc Group, 2005.

[8] About the OSGi Service Platform, Technical Whitepaper. `http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf`, June, 2007.

[9] J. P. Anderson. Computer Security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972.

[10] D. Brewer and M. Nash. The Chinese Wall Security Policy. *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[11] T. Colcombet and P. Fradet. Enforcing Trace Properties by Program Transformation. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, 2000.

[12] J. Dehlinger and N. V. Subramanian. Architecting Secure Software Systems Using an Aspect-Oriented Approach: A Survey of Current Research. Technical report, Iowa State University, 2006.

[13] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of AOSD '04*, pages 141–150, USA, 2004. ACM.

[14] G. Edjlali, A. Acharya, and V. Chaudhary. History-based Access Control for Mobile Code. In *Proceedings of CCS '98*, pages 38–48, New York, USA, 1998. ACM.

[15] U. Erlingsson. *The Inline Reference Monitors Approach to Security Policy Enforcement*. PhD thesis, Cornell, 2004.

[16] F.B.Schneider, G.Morrisett, and R.Harper. A Language-based Approach to Security. In *Informatics 10 Years Back, 10 Years Ahead, LNCS 2000*, pages 86–101, 2000.

[17] C.-C. Huang, P.-C. Wang, and T.-W. Hou. Advanced OSGi Security Layer. In *Proceedings of AINAW '07*, pages 518–523, USA, 2007. IEEE Computer Society.

[18] P. Hui and J. Riely. Temporal Aspects as Security Automata. In *Proceedings of FOAL' 06)*, pages 19–28, 2006.

[19] T. Keuler and Y. Kornev. A Light-weight Load-time Weaving Approach for OSGi. In *Proceedings of Next Generation Aspect Oriented Middleware Workshop, in conjuction with AOSD '08*, 2008.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[21] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[22] J. A. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, 2006.

[23] H.-Y. Lim, Y.-G. Kim, C.-J. Moon, and D.-K. Baik. Bundle Authentication and Authorization Using XML Security in the OSGi Service Platform. In *Proceedings of ICIS '05*, pages 502–507, Washington, DC, USA, 2005. IEEE Computer Society.

[24] P. Parrend and S. Frenot. Supporting the Secure Deployment of OSGi Bundles. In *Proceedings of WoWMoM 2007*. IEEE Computer Society, June 2007.

[25] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[26] F. B. Schneider, D. Kozen, G. Morrisett, and A. Myers. Language-Based Security for Malicious Mobile Code. Technical report, Cornell Univ Ithaca NY, 2003.