ACIR: An Aspect-Connector for Intrusion Response

Mohammad Gias Uddin[1], Hossain Shahriar[2], and Mohammad Zulkernine[2]
[1]Department of Electrical and Computer Engineering, [2]School of Computing
Queen's University, Kingston, Ontario, Canada K7L 3N6
{gias, shahriar, mzulker}@cs.queensu.ca

Abstract

*The modularization concept behind component-based software (CBS) cannot be applied effectively for cross-cutting concerns such as security. Aspect-oriented programming (AOP) helps in better modularization by identifying cross-cutting concerns and providing a suitable way to separate those concerns. In this paper, we provide an aspect-connector based intrusion response (detection and prevention) architecture for CBS by bringing the concepts of aspects into components. The aspect-connector is named as ACIR (**A**spect **C**onnector for **I**ntrusion **R**esponse). Component interfaces act as join points, and aspects containing pointcuts and advices are defined in ACIR configuration file. Advices applicable to particular pointcuts are two types. Signature advices are used to detect intrusions, and action advices are executed to prevent intrusions. A prototype of this architecture is implemented and evaluated using some intrusions included in the Web Application Security Consortium (WASC) intrusion list. This approach detects and prevents intrusions in CBS while maintaining encapsulation, reusability, and modularity.*

1 Introduction

Cross-cutting concerns are not localized in one single module as they are called from different modules (scattered code), or they are tangled/glued to the main functionality of a module (tangled code) [3]. Security functionalities are tangled with the functional codes of a software, and are treated as additional functionalities [13]. Since security is a cross-cutting concern, it is difficult to handle properly in Component-Based Software (CBS), where modularization, reusability and independence of components are primary concerns. Component models have components, interfaces, and connectors [14]. A component publishes its services in its interfaces, and connectors are used to bind components based on interfaces. Aspect-Oriented Programming (AOP) helps in separating crosscutting concerns in "aspects" by introducing the concepts of join point, pointcut, and advice [3]. An aspect contains a set of pointcuts and advices. For CBS, join points are component services published in the interface, and a pointcut is a set of join points. AOP concepts can be used in CBS by two ways [2]: aspect-component and aspect-connector. An aspect-component contains aspects in the component-level, where an aspect-connector contains some aspectual behavior in its configuration file. The aspectual behavior includes aspect definition in the form of pointcut and advice. An aspect-connector enforces the application of advices on particular join points according to aspect definitions in its configuration file. The use of aspect-component prevents adding or modifying pointcuts and advices after the system deployment. However, a suitable implementation of aspect-connector might overcome this difficulty. Since intrusion response is a cross-cutting concern and the nature of intrusions evolves over time, this response approach could be incorporated in aspect-connector in such a way that updating of such concerns could be applied without updating the whole system.

In this paper, an aspect-oriented approach to intrusion response (detection and prevention) is proposed for CBS considering components as black boxes. An intrusion response architecture is presented which provides intrusion response facilities to components by separating intrusion response concerns in an aspect-connector. The aspect-connector is named as ACIR (**A**spect **C**onnector for **I**ntrusion **R**esponse). In this architecture, components are deployed in a container, where the container maintains communication among components. The container uses ACIR, if intrusion response facilities are required for particular component services. Aspects (pointcuts are advices) are defined in ACIR configuration, and executable advices are stored in repositories outside the connector. A detection advice detects intrusions on particular poincut, where a prevention advice prevents the intrusion. An intrusion response algorithm employed by ACIR is provided. A prototype is implemented and evaluated using some intrusions included in the Web Application Security Consortium (WASC) [6] intrusion list. The contributions of this work are twofold.

First, an aspect-connector based intrusion response architecture for CBS is proposed by presenting the definitions of aspects. Second, an intrusion response algorithm is provided based on the architecture. The major implication of this work is that it maintains encapsulation, modularization, and reusability of components.

The remainder of this paper is organized as follows. Section 2 discusses the related work. The intrusion response approach is presented in Section 3. Section 4 discusses the experimental evaluation, while Section 5 concludes the work and provides a number of future research directions.

2 Related Work

We are not aware of any other intrusion response approach for CBS using AOP. However, our approach is motivated by several other work concerning component security, AOP and CBS. Ren et al. [1] propose a connector-centric approach to handle component security. This connector works as a middleware between the clients and the server. Their work focuses on the client-server authentication to provide secure communication, which is not related to intrusion response. Song et al. [5] provide security policies, which are weaved dynamically into components for secure web-service composition. The weaved security policies do not specify how intrusions against the services can be detected and prevented. Pawlak et al. [12] propose JAC (Java Aspect Component) framework for building aspect-oriented distributed applications in Java. They propose the concept of Aspect Components (ACs) which are dynamic as they can be added, removed, or controlled in run-time. ACs provide the definition of distributed pointcuts. In our work, we have defined the aspects in the connector level with necessary modifications. The aspect definition in our approach is not hard-coded, instead it is specified in the descriptor file of the ACIR based on the join point declarations in the component descriptor files. Aldrich [4] proposes Open Modules to ensure encapsulation and openness of the components while using AOP. We have maintained the Open Modules rules in our architecture. While Aldrich uses the definition of pointcut in the interface level of a component, we use the definition of pointcuts in connector level.

3 The Intrusion Response Approach

In this section, the intrusion response approach is discussed by providing the response architecture in Section 3.1 and the algorithm in Section 3.2. The approach is elaborated using the Brute Force attack [6]. The Brute Force attack is executed by a malicious user against a login server. To login, a user provides user name (`userID`) and password (`passWD`). A malicious component can attempt by trial and error process to guess the user name and password of a legitimate user. The intrusion is detected when at least N (e.g., 40) failure login attempts come from the same malicious component, where the interval between each login attempt is less than t (e.g., 4) time units. The response to this intrusion can be handled in two ways for CBS. First, the response concern can be implemented in a separate component, and the login server can provide some tangled code with its login functionality to invoke the response concern. Second, the response concern can be incorporated to the login functionality as an additional functionality. In both cases, this intrusion response concern is tangled to the login functionality. Therefore, we should separate this Brute Force response concern in an aspect.

3.1 The Intrusion Response Architecture

The intrusion response architecture is presented in Figure 1. Components are deployed in a container, and communications among components are maintained by Event Dispatcher. The published services in a component interface are listed in `ComponentDescriptor.xml` file. Figure 2 presents a snippet of `ComponentDescriptor.xml` by showing the login service provided by component1, where the service name is `getLogin`, and class name is `ServiceLogin`. The parameters for this service are `userID`, `passWD`, and `HttpRequest`, and the return type is `HttpResponse`. Upon the request of a service from a component, the Event Dispatcher module uses the descriptor file to determine the component providing the requested service. The aspect-connector, ACIR has a configuration file, `ACIR.xml` to define aspects based on the services in the component interfaces. The snippet of `ACIR.xml` is provided in Figure 3, where the provided aspect (`loginIntrusions`) contains pointcut and advice definitions for intrusions against login service. The aspect `loginIntrusions` contains possible intrusion names for the targets specified in the pointcuts. For login services, two types of intrusions can happen: the Brute Force attack and the SQL Injection attack. The translation of this pointcut captures all login services:

```
pointcut pcBF(): target(component1.ServiceLogin) &&
call(public +.*Login*(...))
&& args(String, String, HttpResponse)
```

For the Brute Force attack, the detection advice is executed after the login service is invoked. The prevention advice is executed around the login service upon the detection of the Brute Force attack. The intrusion detection advices are called as signatures, and the prevention advices are called as actions. The Brute Force signature uses signature attributes, `bfAttr` from the `SignatureAttributes` configuration file for detection purpose. Figure 4 shows that `bfAttr` has four detection attributes, `Match`, `Interval`,

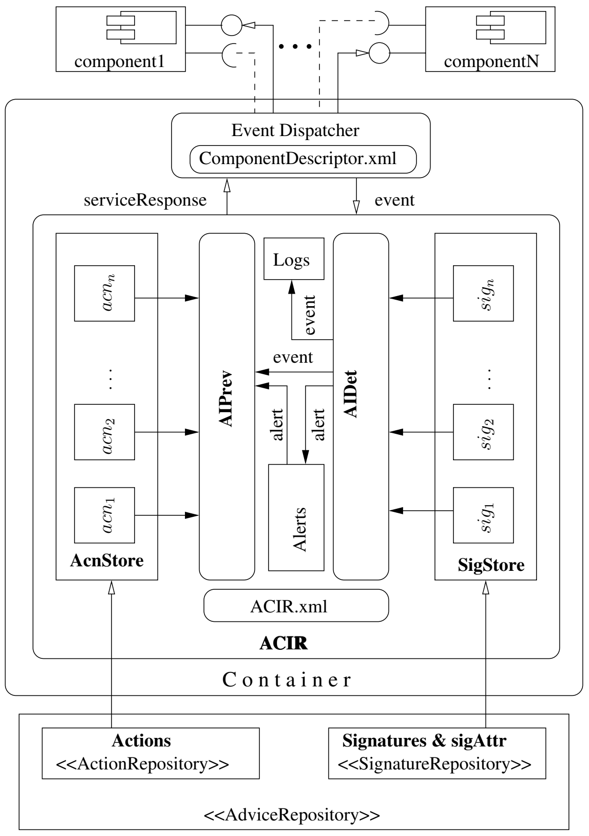


Figure 1. The intrusion response architecture

```
<Component name="component1">
    <ProvidedServices>
        <ProvidedService provides="login"
         Service= "getLogin"
         ret-type="HttpResponse"
         param="String, String, HttpRequest"
         class="ServiceLogin">
        </ProvidedService>
        ...
    </ProvidedServices>
</Component>
```

Figure 2. A snippet showing login service publish in `ComponentDescriptor.xml`

`Frequency`, and `HttpResponse`, where `Match` specifies that session id from the same requestor (`rID`) should be checked. The detection advice considers a login request as suspicious if the request fails, i.e., if the `HttpResponse` is a failure.

The Event Dispatcher module explores the aspect definitions in `ACIR.xml` to check whether the service is used as a target in any pointcut definition of an aspect. If any aspect

```
<ACIR>
      <aspect="loginIntrusions">
        <pointcuts>
            <pointcut="pcBF"
            target="component1.ServiceLogin"
            call= "public +.*Login*(..)"
            args="String, String, HttpRequest">
            </pointcut>
            <possibleIntrusions="BruteForce, SQLInjection">
            </possibleIntrusions>
        </pointcuts>
        ...
        <advices>
            <advice>
                <type="after" pointcut="pcBF"
                 purpose="detection"
                 intrusion="BruteForce">
                </type>
                <class id="BruteForceDetectAdvice"
                sigAttr="bfAttr" param="event, intrusion">
                </class>
            </advice>
            <advice>
                <type="around" pointcut="pcBF"
                 purpose="prevention"
                 intrusion="BruteForce">
                </type>
                <class id="BruteForcePreventAdvice"
                param="event, intrusion">
                </class>
            </advice>
            ...
        </advices>
      </aspect>
        ...
</ACIR>
```

Figure 3. A snippet of the `ACIR.xml` **defining aspects for login type intrusions**

```
<SignaureAttributes>
    <SigAttr="bfAttr" Match="SessionID"
     Interval="4" Frequency="40" HttpResponse="Failure">
    </SigAttr>
    ...
<SignaureAttributes>
```

Figure 4. Signature attributes of the Brute Force attack

definition includes the service as a possible join point, the request handling is delegated to ACIR as an event.

The detection of intrusion in ACIR is handled by AIDet module which takes events from Event Dispatcher module and uses signatures on the particular poinctcut specific join points. AIDet uses the Logs database to store temporary information related to an intrusion. Upon the detection of an intrusion, AIDet generates alerts in the Alerts database, and passes the event to the AIPrev module. AIPrev executes the corresponding action to prevent the intrusion. SigStore loads each detection advices, i.e., signatures as a separate entity such as sig_1, sig_2, ..., sig_n. Prevention advices, i.e., actions are loaded in AcnStore. Each action is loaded as a separate entity such as acn_1, acn_2, ...,

acn_n. For example, the Brute Force attack detection class is `BruteForceDetectAdvice` and prevention class is `BruteForcePreventAdvice`. This advice may contain the action for terminating the login session between the client and server components (e.g., `terminateSession (Provider, Requestor)`). The advices are stored in `AdviceRepository`, which is outside the container. The separation of the advice repository from the container helps in adding or modifying advices even after the components are deployed.

3.2 The Intrusion Response Algorithm

The algorithm is provided in Listing 1, where signatures and actions are loaded in $SigStore$ and $AcnStore$ respectively (Lines 01-06). For a service request r, the requestor id rID, service name sn, request arguments rp (reqArgs), and the service provider id (p) are retrieved (Lines 07-08). Based on r, the `ACIR.xml` file is explored for possible pointcut (pc), and possible intrusion names (IN) (Lines 09-10). IN lists possible intrusion names for pc. An event e is created by Event Dispatcher using the retrieved information and the service request time ($curTime\ t$) (Line 11). The detection is performed by `detectIntrusion(`e`, `$IN[i]$`)` in AIDet (Line 13), where $IN[i]$ represents an intrusion name. If the corresponding intrusion is detected, an alert is generated (Line 14). All alerts are stored in the Alerts database I. The AIPrev module matches each event e with each alert i in I, and if the event is matched with an alert for possible intrusion $intr$, the intrusion is prevented using prevention advices (Lines 16-18). `detectIntrusion(`e`, `$intr$`)` invokes corresponding signature to detect intrusion (Lines 20-24), where `matchWithAlert(`e`, `$intr$`)` matches an event with the logged alerts for the specific intrusion (Lines 25-30). Intrusion is prevented by invoking the corresponding action in `preventIntrusion(`e`, `$intr$`)` (Lines 31-33).

The Brute Force attack is executed against a login request r which is requested by `componentN` at time t_1. According to `ComponentDescriptor.xml`, the provider is `component1`, and the service name is `getLogin`. The pointcut definition of aspect `loginIntrusions` in `ACIR.xml` specifies two possible intrusions against r, where one of them is the Brute Force attack. The event e has the form $<login$, $rID_{componentN}$, $p_{component1}$, "$userID; passWD$", $t_1>$, where $login$ is the service name, $rID_{componentN}$ is the service requestor id, $p_{component1}$ is the service provider id, "$userID; passWD$" is the string containing the request arguments, t_1 is the service request time. The Brute Force detection advice is executed based on e after the login service is invoked. If the intrusion $BruteForce$ is found at time t_1, an alert i is generated in the form $<login$, $BruteForce$, $rID_{componentN}$, $p_{component1}$, $t_1>$ in the Alerts database (I). AIprev matches e with i and finds that a Brute Force attack is detected based on e. AIPrev then executes the Brute Force prevention advice around the login service, e.g., by terminating login request between the provider and the requestor to stop further invocation of the service by the requestor.

Listing 1: The Intrusion Response Algorithm

Input: A set of n intrusion signatures (S) and actions (A), and the requested service r.
Output: If an intrusion is found, the corresponding action a is executed.

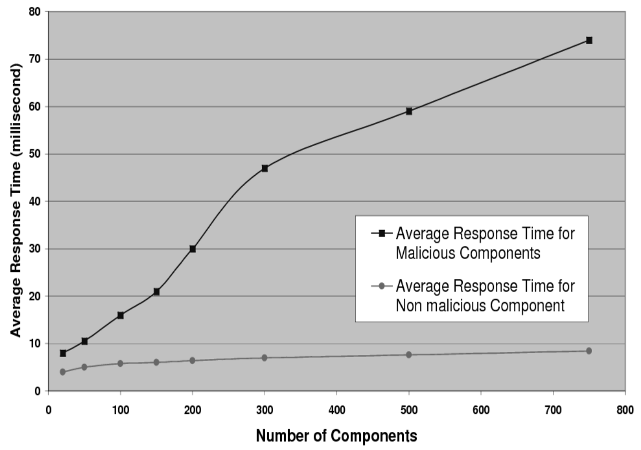
```
00. AOIR (Signatures $S$, Actions $A$, serviceRequest $r$)
01. FOR each intrusion signature $s \epsilon S$ DO
02.   SigStore:= SigStore $\cup$ $s$
03. END FOR
04. FOR each intrusion action $a \epsilon A$ DO
05.   AcnStore:= AcnStore $\cup$ $a$
06. END FOR
07. Get requestor $rID$, serviceName $sn$, and reqArgs $rp$ from $r$
08. Find serviceProvider $p$ from ComponentDescriptor.xml
09. Find the pointcut $pc$ related to $sn$ from ACIR.xml
10. Get possible intrusionNames $IN$ based on $pc$ from ACIR.xml
11. Create event $e$ using $rID$, $sn$, $p$, $rp$, and curTime $t$
12. FOR ($i = 0; i < IN.length; i++$) DO
13.   IF (detectIntrusion ($e$, $IN[i]$) = TRUE) THEN
14.     Generate alert $i$ in $I$
15.   END IF
16.   IF (matchWithAlert ($e$, $IN[i]$) = TRUE) THEN
17.     preventIntrusion ($e$, $IN[i]$)
18.   END IF
19. END FOR
20. detectIntrusion (event $e$, string $intr$)
21.   Invoke $intr$ detection advice $s$ from ACIR.xml based on $e$
22.   IF the intrusion is detected THEN Return TRUE
23.   ELSE Return FALSE
24.   END IF
25. matchWithAlert (event $e$, string $intr$)
26.   FOR each alert $i \epsilon I$ DO
27.     IF ($e.p$ = $i.Victim$ && $e.sn$ = $i.Service$
          && $e.rID$=$i.Requestor$ && $intr$ = $i.intrusion$
          &&$e.curTime$=$i.detectedTime$) THEN Return TRUE
28.     ELSE Return FALSE
29.     END IF
30.   END FOR
31. preventIntrusion(event $e$, string $intr$)
32.   Retrieve prevention advice $a$ using ACIR.xml
33.   Invoke advice $a$ based on $e$ to prevent $intr$
```

4 Experimental Evaluation

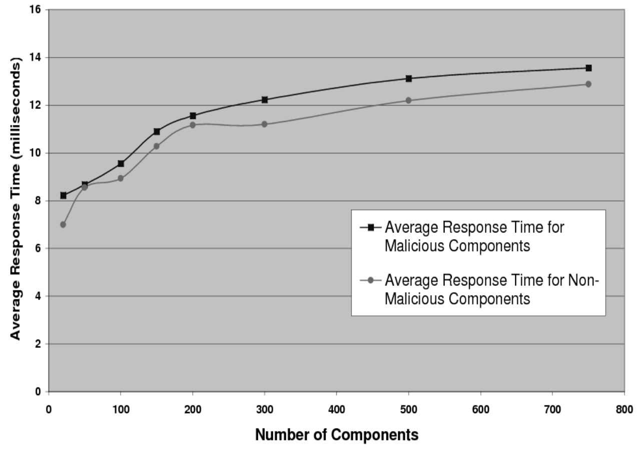
A prototype of the system is implemented using Java, where a container is developed by maintaining the EJB restrictions [7]. The developed container uses java reflection mechanism to realize component services. The container includes the ACIR connector to provide intrusion response facilities to the components. We consider that repositories are used for storage purpose, and they can be modified without recompiling the system [8].

We have evaluated the prototype using three categories of intrusions/attacks chosen from the Web Application Security Consortium (WASC) [6] intrusion list: the Brute Force attack, the Session hijacking attack, and the SQL injection attack to cover authentication, authorization, confidentiality and integrity violations. These three categories of violations are the most prevalent in CBS [9, 10]. A component needs authentication service to check the identity of a user, while authorization is required to check whether a user is legitimate to do a particular action [11]. A component providing database access needs to ensure that user data is not exposed to or modified by an intruder.

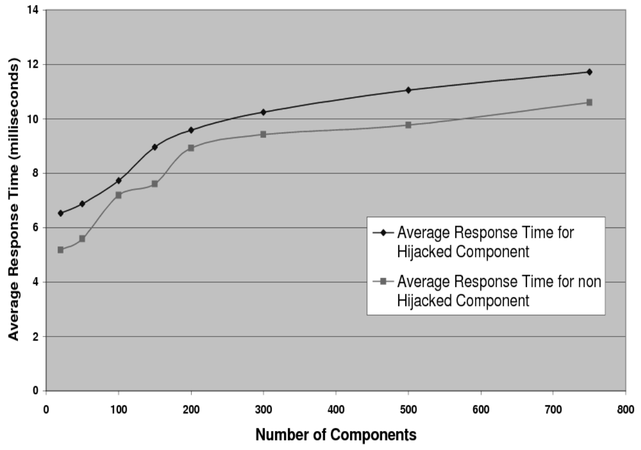
During the experimentation, we have measured the performance of ACIR to observe the effectiveness and efficacy of ACIR in a real-time environment. The criteria is how much time ACIR takes to respond to a request. The results in Figure 5 reveal that ACIR service response times vary based on different attacks. For the Brute Force attack, we assume that half of the components are malicious which use fake information (e.g., fake `userID` and `passWD`). An automated script is run to create N client component instances. A client component is a simple component which gains access to the login server by providing login information. Our architecture allows multiple instances of ACIR based on service requests. However, the Logs database and the Alerts database are shared among the ACIR instances. An event containing fake information is considered as a suspicious event, which for the detection purpose, is logged in the Logs database. The Alerts database also needs to be updated upon the detection of an intrusion. However, an event containing legitimate information is considered as a good event, and it is not logged in the Logs database. Therefore, the response times are almost constant for the non-malicious components. However, with the increase of malicious components, the service response times from ACIR also increase because of the overhead of writing the Logs and Alerts database. The situation changes for the SQL Injection attack and the Session Hijacking attack. In the SQL Injection attack, the user inputs maliciously crafted SQL statements in the login or search field to gain illegal access to system records. In the Session Hijacking attack, a malicious user hijacks the session id of another user to gain unauthorized access to a system. For both of these attacks, the suspicious events do not need to be logged as the states of the attacks can be captured in a single event. However, alerts are needed to be generated upon the detection of the SQL Injection or the Session Hijacking attack. Therefore, there are negligible differences between responding to



(a) The Brute Force Attack



(b) The SQL Injection Attack



(c) The Session Hijacking Attack

Figure 5. Variations in service response time of ACIR based on different attacks

malicious and non-malicious component requests based on the two attacks. According to the experimental results, it is confirmed that the response times of ACIR vary based on different attacks.

5 Conclusions and Future Work

In this work, an aspect-oriented intrusion response (detection and prevention) approach is presented for CBS considering components as black boxes. Intrusion response facilities to the CBS are provided by an aspect-connector named as ACIR. ACIR provides definitions of aspects (pointcuts and advices) for intrusion response. An intrusion response algorithm employed by ACIR is provided, and the response process is elaborated using the Brute Force attack example. A prototype of the framework is implemented and evaluated using the intrusions included in the Web Application Security Consortium (WASC) intrusion list [6]. This work contributes to the automatic intrusion response of CBS and helps in better modularization, reusability, and encapsulation of components.

The intrusion response approach, however, increases the overhead of the system by delegating the intrusion response functionalities in the connector level. For large component-based systems having numerous interactions, the proposed approach may affect to the total system performance. Moreover, the current implementation does not consider the security of the repositories, the Logs and the Alerts database, and the container itself. The prototype will be evaluated by more intrusions targeted for CBS. The intrusion response algorithm will be modified to overcome the overheads. Moreover, ensuring security for the aspect-connector and the repositories will be explored to make the approach more dependable and intrusion-tolerant.

Acknowledgement

This research is partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] J. Ren, B. Taylor, P. Dourish, D. Redmiles. Towards an architectural treatment of software security: A connector-centric approach, in *Proceedings of the 1st Workshop on Software Engineering for Secure Systems (SESS'05)*, St. Louis, Missouri, May 2005. ACM Press: New York, 2005; 1–7.

[2] M. Bynens, W. Joosen, K. Leuven. On the benefits of using aspect technology in component-oriented architectures. in *Proceedings of the 11th International Workshop on Component-Oriented Programming (WCOP'06)*, Nantes, France, July 2006.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-oriented programming, in *Proceedings of The 11th European Conference on Object-Oriented Programming (LNCS 1241)*, Springer-Verlag: Finland, 1997, 220–242.

[4] J. Aldrich. Open Modules: Reconciling extensibility and information hiding, in *Proceedings of The 3rd AOSD workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT'04)*, Lancaster, UK, March, 2004.

[5] H. Song, Y. Sun, Y. Yin, S. Zheng. Dynamic Weaving of Security Aspects in Service Composition, in *Proceedings of the Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06)*, Shanghai, China, October 2006. IEEE CS Press: Washington DC, USA, 2006; 189–196.

[6] Web Application Security Consortium. Web application security consortium: Threat classification V1.0, in `http://www.webappsec.org/projects/threat/` (Accessed in November 2006).

[7] Java^{TM} BluePrints: EJB Restrictions. in `http://java.sun.com/blueprints/qanda/ejb_tier/restrictions.html`, (Accessed in January 2007)

[8] T. Loomis. A java database framework for SQL anywhere, in `http://www.ianywhere.com/downloads/whitepapers/` (Accessed in November 2006).

[9] N. Nissanke. Component security - issues and an approach, in *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Edinburgh, Scotland, July 2005. IEEE Computer Society Press: Los Alamitos, CA, USA, 2005; 152–155.

[10] U. Lindqvist, E. Jonsson. A Map of Security Risks Associated with Using COTS, in *Computer: IEEE*, 1998, **31**(6):60–66.

[11] Q. Zhong, N. Edwards. Security Control for COTS Components, in *Computer: IEEE*, 1998, **31**(6):67–73.

[12] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Aubry, L. Martelli. JAC: An aspect-based distributed dynamic framework, in *Software: Practice and Experience: Wiley*, 2004, **34**(12):1119–1148.

[13] I. Welch, R. Stroud. Re-engineering security as a crosscutting concern, in *The Computer Journal*, 2003, **46**(5):578–589.

[14] C. Szyperski. Component software: Beyond object-oriented programming, in ACM Press and Addison-Wesley, New York, NY, 1998.