A FRAMEWORK FOR PROTECTING EJB APPLICATIONS FROM MALICIOUS COMPONENTS

Hieu Dinh Vo and Masato Suzuki
School of Information Science, Japan Advanced Institute of Science and Technology, Ishikawa, Japan

Keywords: J2EE, EJB, component-based, security, business functions.

Abstract: Enterprise JavaBeans (EJB) components in an EJB application can be obtained from various sources. These components may be in-house developed or bought from other vendors. In the latter case, the source code of the components is usually not available to application developers. The result is that the application may contain malicious components. We propose a framework called BFSec that protects EJB applications from vicious components. The framework examines bean methods invoked by each thread in applications and compares them with pre-defined business functions to check whether the latest calls of threads are proper. Unexpected calls, which are considered to be made by malicious components, will be blocked.

1 INTRODUCTION

Recently, large-scale information systems are built based mainly on software component technology. Besides the benefits such as reducing complexity, time, and the development cost of systems, using components in developing information systems may introduce new security risks. The main source of security issues in component-based systems is that components used in a particular system may come from various sources and that their source code may not be available. This leads to difficulties in assessing the security aspect of the used components. The consequence is the possible introduction of malicious components, which can cause the whole system to be insecure.

EJB (Sun, 2005) is one of the leading technologies for developing component-based applications. In the current EJB applications, the protection of system resources such as files, memory, and network from malicious components is mainly based on the Java protection mechanism (Gong, 2002). In addition, with the use of the role-based access control mechanism, beans are protected from unauthorized users (Sun, 2005). However, in the context of component-based software, we also need another kind of protection: that is the protection of beans from malicious beans.

In this paper, we present our work on building a framework for protecting EJB applications from the malicious beans. We utilize the concept of business function (Vo and Suzuki, 2007) along with the Intercepting Filter pattern (Alur et al., 2003) to ensure that methods of beans are invoked not only by the right person but also from the right places.

In the following section, we describe an example that is used for discussion in later sections. Then, in Section 3, we show that the current mechanism is not sufficient to protect applications from malicious beans. Our framework is described in Section 4. Section 5 presents experimental results and discusses points related to the approach used in the framework. Works related to our research are summarized in Section 6.

2 MOTIVATING EXAMPLE

Our example is the business tier of a virtual online banking system named eBank. The system is developed based on the J2EE platform. Customers of the bank use the system (via Web browsers) to transfer money between accounts. EJB components used for this functionality are shown in Figure 1.

The process of transferring money is initiated by invoking method `transfer()` of `TxBean`. This method will first call `CheckBean.check()` to assess conditions of making the transaction. If there is no problem, `TxBean.transfer()` calls `DoTxBean.doTx()`, in which the actual transaction is done. Finally, method `doTx()` invokes

Dinh Vo H. and Suzuki M. (2008).
A FRAMEWORK FOR PROTECTING EJB APPLICATIONS FROM MALICIOUS COMPONENTS.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - DISI*, pages 264-269
DOI: 10.5220/0001704502640269

`LogBean.log()` for recording information about the transaction.

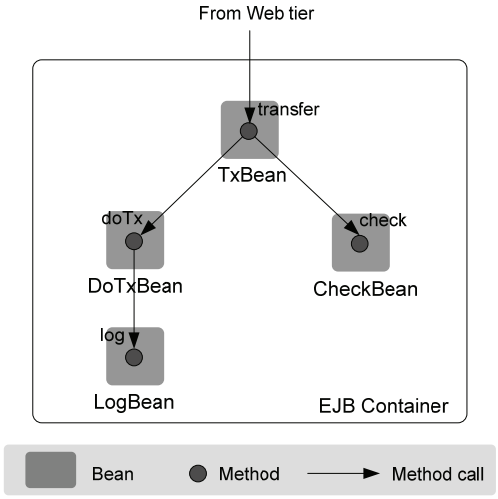

Figure 1: EJB components in the eBank system.

For the sake of simplicity, we only consider the role *customer* of the system, and this role can transfer money between accounts.

3 THE PROBLEMS

Current EJB applications use the configuration file `ejb-jar.xml` to specify several aspects of the applications, such as the dependencies between beans, transactions, and security policy. The access control policy of bean methods is declared within the elements `<method-permission>` in this file.

```
<method-permission>
   <role-name>customer</role-name>
   <method>
    <ejb-name>DoTxBean</ejb-name>
    <method-name>doTx</method-name>
   </method>
</method-permission>
```

Listing 1: Access control policy for a method.

Listing 1 states that the role `customer` can invoke method `DoTxBean.doTx()`. Note that if we allow a role to execute a method *m*, unless we use principal delegation, we must allow that role to invoke all methods called by *m*. If not, the invocation of *m* will not be completed.

In the current EJB access control, we only state which role can execute which method. There is no constraint about where a method might be invoked. This, together with the fact that the source code of beans is not always available to the application developers, may lead to security issues. For example, in our eBank system, we assume that the source code of bean `TxBean` is not available. In order to allow the role `customer` to transfer money, we must allow this role to execute `TxBean.transfer()`, `DoTxBean.doTx()`, `CheckBean.check()`, and `LogBean.log()`. However, the policy does not point out where these methods can be invoked from. If the `TxBean` is malicious (e.g. the bean itself is malicious or it is being controlled by a malicious person), it can invoke `LogBean.log()` directly (on behalf of the `customer` role) without going through `DoTxBean.doTx()`. This is unexpected.

In addition, `TxBean.transfer()` is supposed to execute `CheckBean.check()` before invoking `DoTxBean.doTx()`. At runtime, we cannot ensure that, in every case, `TxBean` will invoke `CheckBean.check()` before invoking `DoTxBean.doTx()`.

The above analysis shows that, in the context of component-based software development, we need a stricter way of specifying and enforcing access control policy than the current approach. In the next sections, we describe our BFSec framework, which aims at achieving this.

4 BFSEC FRAMEWORK

The core idea of the approach is, first, at design time, to define call flows (i.e. series of interactions between methods) that may happen in that application. Then, at runtime, we check all method invocations to ensure that they conform to the defined call flows. A method invocation is allowed only if the invocation belongs to at least one of the defined call flows.

In our framework, we use the concept of *business functions* (Vo and Suzuki, 2007) to define the call flows (at design time). Meanwhile, at runtime, a call flow is a series of methods invoked by a *thread*. The framework ensures that, at any point during the lifetime of a given thread, the thread must conform to at least one of the defined business functions. We enforce this policy by assigning a set of potential business functions to each thread the first time the thread invokes a method. This set is then updated each time the thread invokes a method. The framework blocks any thread having no business function associated with it.

We equip beans with interceptors to update business functions of threads. In addition, the framework uses other modules for providing information about defined business functions and

call flow at runtime. Figure 2 shows the overall architecture of the framework.

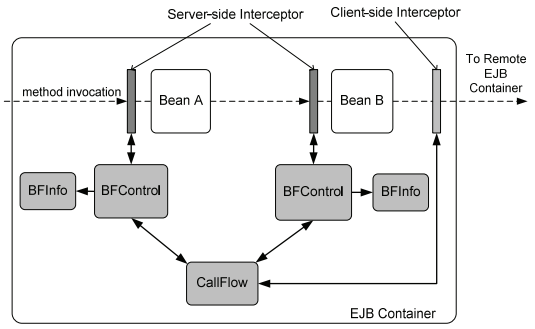

Figure 2: Architecture of BFSec framework.

The BFInfo module contains information of business functions related to the corresponding bean. This information is extracted from business functions defined by application developers. The CallFlow module manages information of all threads in an EJB container. There are many threads in a system, but the CallFlow module only concerns threads that invoke at least one bean method. Only one CallFlow module is used for an EJB container. The BFControl modules are responsible for policy enforcement. For each bean, we provide a BFControl module. This module uses information obtained from BFInfo (about business functions) and from CallFlow (about threads) to update the set of business functions associated with threads and to decide whether calls are permitted or not. In our system, we use two kinds of interceptor including server-side interceptors and client-side interceptors. The server-side interceptors are used by BFControl to update business functions of threads. The client-side interceptors are used when invoking methods belonging to beans in different machines. In this literature, if a stand-alone word "*interceptors*" is used, we mean the server-side interceptors.

The next sections present the details of the framework.

4.1 Business Functions

The concept of business function is introduced in the work about a flexible approach for specifying access control in EJB applications (Vo and Suzuki, 2007). In this paper, we revise the description of business function so that it is suitable for the BFSec framework.

We use the XML format to describe business functions. The Listing 2 is the Document Type Definition (DTD) for a business function description.

```
<?xml version= "1.0"?>
<!ELEMENT business-function (invoke)>
<!ALIST business-function name CDATA>
<!ELEMENT invoke (method, invoke*,
block*)>
<!ELEMENT method CDATA>
<!ELEMENT block (invoke*, block*)
<!ALIST block type (if | dowhile |
switch)
```

Listing 2: DTD of business function description

A description of a business function starts with the element `<business-function>` and the name of that business function, then followed by a description of interactions between methods. A method invocation is described by element `<invoke>`, which includes the name of the invoking method. An `<invoke>` element may contain other `<invoke>` elements to describe a method which calls other methods. The method invoked first in a business function is the *entry method* of that business function. We also use the element `<block>` to group invocations and to provide flow control. The execution of a block can be modified by using the attribute `type` of the `<block>` element. The values of this attribute include `"if"`, `"switch"`, `"dowhile"`, whose semantics are similar to that used in Java programming language. In descriptions of business functions, we do not consider the conditions that control invocations. For example, if `type="if"`, we do not care about when the methods inside the `<block>` will be executed. The reason for this is that the condition is a part of the internal state of the bean, so we should not take it into account.

4.2 Information Extracted from Business Functions

From business function descriptions, we can extract information that is used for updating business functions associated with each thread at runtime. The following variables and functions describe what kind of information we extract from defined business functions.

- R: set of roles. B: set of business functions of the application. M: set of methods involving at least one of the business functions in B. r: $\in R$, is the role of the principal who is making the call. *caller, callee,* and *preMethod*: $\in M$; *caller* is the method making the call, *callee* is the called method. *preMethod* is the method (if it exists) invoked by *caller* before invoking *callee*. *bfs:* $\subset B$, a set of business functions

associated with a certain thread. *bf*: $\in B$, a business function.

- *allow(r,bf):* returns *true* if users in role *r* are allowed to invoke business function *bf*. Otherwise, the function returns *false*.
- *allow(caller, callee, bf)*: this function returns *true* if in the business function *bf*, *caller* invokes *callee*. Otherwise, the function returns *false*.
- *entry(bf)*: this function returns the entry method of business function *bf*.
- *preCall(caller, callee, bf)*: This function returns a set of methods so that if *m* is an element of this set then *caller* calls *m* before invoking *callee* and the business function containing these invocations is *bf*.
- *initBf(r, callee)*: *callee* $\in$ *M* is the called method. This function returns a set of business functions so that if *bf* is an element of this set then *entry(bf)==callee* and *allow(r,bf)==true*.
- *nextBf(bfs, caller, callee, preMethod)*: *bfs* $\subset B$, is the set of business functions this thread may involve. This function returns a subset of *bfs* so that if *bf* is an element in this subset, then *allow(caller, caller, bf)==true* and *preMethod* $\in$ *preCall(caller, callee, bf)*.

In the BFSec framework, the above functions are provided by the BFInfo module. The BFControl module uses these functions to update the set of business functions for each thread at runtime. However, in order to use these functions, we need to know *callee*, *caller*, and *preMethod*. The next section describes how to obtain these values at runtime.

4.3 Obtaining Runtime Information of Threads

In this section we describe how to obtain information that is used for updating sets of business functions associated with threads. We also explain the way in which this information is managed by the CallFlow module.

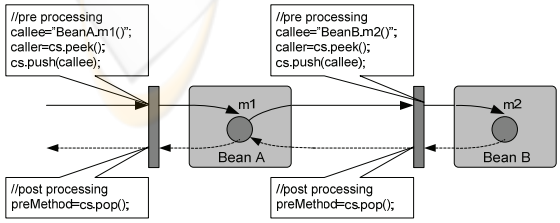

Figure 3: The solution for determining the callers.

The information about a thread that we need to know is the callee, the caller, and the previous called method of the caller. For the callee, we can easily acquire because at interceptors we know which method we are invoking. However, determining the caller and the previous method called by the caller is quite difficult. The solution to this problem is to equip a stack for each thread. Figure 3 illustrates our solution. In the figure, the stack used for the thread is `cs`. The operation on the stack includes `peek()`: returns the value on top of the stack; `push(m)`: pushes m onto the top of the stack; `pop()`: removes and returns the top value of the stack.

For each thread in an application, we need to maintain: `bfs`: a set of business functions that is associated with the thread, `cs`: a stack containing methods the thread has been invoking, and `preMethod`: the method last called by the method on the top of the `cs` stack. In our framework, each thread is provided an entry containing this information and these entries are managed by the CallFlow module.

In cases where the caller and the callee are located in different machines that solution will not work because we base it on threads. This is the reason why we use client-side interceptors. The task of these interceptors is to send information of threads making the call (including the set of business functions, the caller, and the previous called of the caller) to the remote system. This means that when checking permission of a remote call, instead of obtaining information of threads from the CallFlow module, the BFControl module will use information sent by the client-side interceptors.

4.4 Business Function Enforcement

This section describes how the BFControl module uses the BFInfor module and the CallFlow module to update sets of business functions associated with threads and to enforce call flows at runtime.

At the interceptors, the set of business functions associated with a thread is updated as follows.

```
if(caller==null)
   bfs=initBf(r,callee)
else
   bfs=nextBf(bfs,caller,callee,
                  preMethod)
if(bfs==null)
   return false;// invocation blocked
else
   return true;// invocation allwed
```

Listing 3: The update of business functions associated with a thread.

Table 1: The overhead introduced by the BFSec framework.

Business Functions	Without BFSec (ms)	With BFSec (ms)	No. inter-bean invocations	% of time consumed by BFSec	Average time per check (ms)
Account History	2.778	2.842	58	2.3	0.0011
ATM Withdraw	0.352	0.386	19	9.7	0.0018
Transfer	0.413	0.432	9	4.5	0.0021
Create Customer	0.201	0.207	3	2.9	0.0019
Create Account	0.255	0.260	4	2.3	0.0012

In the above code segment, we depend on the caller to determine `bfs` of a thread. `caller==null` means that this is the first time the thread invokes a method. In this case, `bfs` is initiated by `initBf(r,callee)`. This function returns a non null set only if `callee` is the entry method of some business functions and if role `r` is allowed to execute these business functions.

When the caller is not null this means that the thread has invoked methods before and the thread is associated with at least one business function (contained in `bfs`). `bfs` is updated by function `nextBf(bfs,caller,callee,preMethod)`.

In both cases (caller is null and not null), `bfs==null` means that there is no business function associated with the thread and the thread will then be blocked.

5 EXPERIMENTAL RESULTS

We have implemented the BFSec framework for JBoss AS 4.0. The application we used for testing is Duke's Bank application (Sun, 2006). The application contains 7 EJB components. We defined 13 business functions such as ATM Withdraw, Account History, and Create Customer. The test was performed on a machine with 2 CPUs AMD 2.4GHz, 4 GB RAM, Fedora Linux 6.0 and Java 1.5. The framework ensures that the security problems described in section 3 cannot happen. In our experiment with Duke's Bank application, every invocation that does not conform to any business function was blocked.

However, the introduction of modules into the original version of the application server should cause overhead. Table 1 shows the results of our experiment about overhead caused by BFSec framework. The results show that the latency caused by the framework is relatively minor and that the time needed for each check is almost the same. This means that for a given business function, the latency caused by the framework increases linearly with the number of inter-bean invocations.

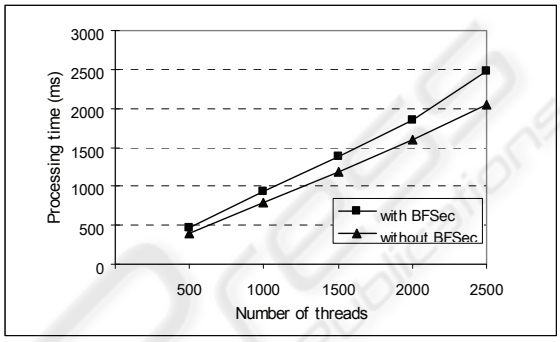

Figure 4: The latency caused by the framework in cases of multi-threaded applications.

In our architecture, there is only one CallFlow module in an EJB container. The consequence is that the module becomes a bottleneck when there is more than one thread in the system. However, because the only operations that needed synchronizing are the adding and deleting an entry to CallFlow (i.e. when a new thread enters the EJB container and when a thread goes out of the EJB container) the latency caused by the synchronization is not significant (Figure 4). In this experiment, we tested the business function Account History with 500 to 2500 threads initiated at once.

6 RELATED WORK

Regardind the protection of EJB components from illegal access, several papers focus on the static analysis of bean code and security policies associated with the applications. This direction includes works of Naumovich and Centonze (Naumovich and Centonze, 2004), Sreedhar (Sreedhar, 2006), and Pistoia et al. (Pistoia et al., 2007) These works either require source code of beans or rely on call graphs produced by byte code analysis tools (but they are not always accurate).

In the current approach of EJB, if a role tries to invoke a method without permission, an exception will be thrown. Evered (Evered, 2003) proposes the

idea that if a role does not have permission to execute a method it should not be aware of the existence of that method.

The above approaches focus on protecting beans from illegal access made by unauthorized people. They cannot protect applications from security issues presented in Section 3 of this paper.

Clarke et al. (Clarke et al., 2003) propose an approach for checking beans at deployment time to make sure that, at runtime, beans are totally controlled by application servers. Their approach can protect beans from access that bypasses the application servers but not the access from beans inside the application servers. Therefore, again, the approach cannot solve the problem presented in the previous section.

7 CONCLUSIONS

In this literature, we have shown that the current approach for protecting EJB applications is not secure enough. We have presented our approach, the BFSec framework, for strengthening security of EJB applications. The idea of the approach is to use the business function concept to define call flows that may happen in an application. After that, at runtime, by collecting information of threads in the application and comparing with the defined business functions, we ensure that threads conform to defined call flows.

One of our future works is to extend the framework so that it can handle threads created by components and by the EJB container. In addition, when the number of EJB components in an application grows, the task of defining business functions should become complicated. We intend to provide a tool that can extract business functions from design documents such as UML diagrams. Our further aim is to build a multi-layer framework for securing EJB applications.

REFERENCES

Alur, D., Malks, D., Crupi, J., Booch, G. & Fowler, M. (2003) *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*, Sun Microsystems, Inc.

Clarke, D., Richmond, M. & Noble, J. (2003) Saving the world from bad beans: deployment-time confinement checking. *SIGPLAN Not.,* 38**,** 374-387.

Evered, M. (2003) Flexible enterprise access control with object-oriented view specification. *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003 - Volume 21.* Adelaide, Australia, Australian Computer Society.

Gong, L. (2002) Java 2 Platform Security Architecture [online]. [Accessed Dec. 2007]. Available from WWW: http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html.

Naumovich, G. & Centonze, P. (2004) Static analysis of role-based access control in J2EE applications. *SIGSOFT Softw. Eng. Notes,* 29**,** 1-10.

Pistoia, M., Fink, S. J., Flynn, R. J. & Yahav, E. (2007) When Role Models Have Flaws: Static Validation of Enterprise Security Policies. *Software Engineering, 2007. ICSE 2007. 29th International Conference on.*

Sreedhar, V. C. (2006) Data-centric security: role analysis and role typestates. *Proceedings of the eleventh ACM symposium on Access control models and technologies.* Lake Tahoe, California, USA, ACM.

Sun (2005) Enterprise JavaBeans version 3.0 [online]. [Accessed: Dec. 2007]. Available from WWW: http://java.sun.com/products/ejb/.

Sun (2006) The J2EE 1.4 Tutorial [online]. [Accessed: Dec. 2007]. Available from WWW: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.

Vo, H. D. & Suzuki, M. (2007) An Approach for Specifying Access Control Policy in J2EE Applications. *14th Asia-Pacific Software Engineering Conference.* Japan, IEEE.