An Approach to Detecting Inter-Session Data Flow Induced by Object Pooling*

Bernhard J. Berger and Karsten Sohr

Center for Computing Technologies (TZI), Universität Bremen
Bibliothekstr.1, 28359 Bremen, Germany
{berber|sohr}@tzi.de

Abstract. Security tools, using static code analysis, are employed to find common bug classes, such as SQL injections and cross-site scripting vulnerabilities. This paper focuses on another bug class that is related to the object-pool pattern, which allows objects to be reused over multiple sessions. We show that the pattern is applied in a wide range of Java Enterprise frameworks and describe the problem of inter-session data flows, which comes along with the pattern. To demonstrate that the problem is relevant, we analyzed different open-source and a proprietary commercial software, with the help of a detection approach we introduce. We were able to show that the problem class occurred in these applications and posed a threat to the confidentiality of the closed-source software.

1 Introduction

Security tools, using static code analysis, are employed by software producers more and more frequently to detect security bugs introduced during the implementation. These static analyses can find bug classes, such as SQL injections and cross-site scripting vulnerabilities. The commercial success of those tools shows that industry is realizing the importance of software security [10].

One kind of applications that are prone to the vulnerabilities mentioned above are business applications, offering services or web interfaces to customers. These systems share the property that they process data from different users at the same time that must be handled confidentially. To separate data from different users, the applications provide a session, an object that represents the server-side state [24]. The common bug classes pose a threat to the confidentiality of the user's data and the availability of the system.

A frequently-used framework to implement business applications is the Java platform, Enterprise Edition that supports different APIs to ease the development of business applications [19]. The core framework is complemented by different libraries like Apache Struts [27] or Spring [26], which provide additional capabilities and should reduce development time.

* This work was supported by the German Federal Ministry of Education and Research (BMBF) under the grant 01IS10015B (ASKS project).

Within this paper, we describe an additional threat to the confidentiality of a user's data that relates to the object-pool pattern described by Kircher and Jain [14]. This pattern is used within different enterprise frameworks to improve the performance by reusing class instances for different requests [25]. Nevertheless, these pooled instances may create the possibility of confidential data-flows between different users of a system if they contain fields that are not handled correctly. This behaviour is known as the "Object Cesspool anti pattern" in the development community. Furthermore, we will show a light-weight and bytecode-based approach to detecting such vulnerabilities. The described approach is evaluated with the help of a closed-source and two open-source systems. We detected possible security vulnerabilities in the analyzed software and the vendor of the commercial software approved that a part of the reported findings were exploitable .

The remainder of the paper is structured as follows. An overview of related work is given in Section 2, followed by a detailed description of the problem, as well as a possible way to address it in Section 3. To show the validity of our approach, we evaluate our proposed solution in Section 4 and conclude with a summary of our work and a look on the next steps we will pursue in the future.

2 Related Work

There is a plethora of works that aim to find security vulnerabilities with the help of static analysis [4, 7, 29, 18]. One approach that handles vulnerabilities for Java enterprise applications was developed by Livshits and Lam [16]. They present a framework to detect different input-related vulnerabilities, such as SQL injections, cross-site scripting, HTTP response splitting, path traversal, and command injections. To find possible vulnerabilities, they identify sources of user input and track it through the entire application to certain methods that are known to be prone to a specific kind of vulnerability. When a possible data flow is found, it is an evidence for a vulnerability and will be reported.

Hammer and Snelting describe how static analysis can be used to check whether the confidentiality or the integrity of the data processed by a program can be threatened by a user [13]. For their information-flow control, they use a program dependence graph, a representation already known in the area of static program analysis [15, 3, 11]. In a succeeding work, Hammer evaluates his approach with some real-world examples and lists results of his performance measurements [12].

Another well-known vulnerability class that can be detected with static analysis is time-of-check-to-time-of-use (TOCTTOU) vulnerabilities [5]. Here, an attacker tries to manipulate a file an application tries to read between the time it checks some properties (e.g. access rights or existence) and the usage of a file. Since the two operations are not atomic, an attacker might manipulate the file between the two operations, such as creating a symbolic link to another file to trick the application into deleting or overwriting it. It is a special kind of race condition between the programmer and the attacker.

3 Inter-Session Data Flow

In the following section, we first summarize whether certain frameworks or their implementation may use the object-pool pattern and how to detect these pooled objects. Afterwards, we give a detailed description of the problem and complement it with an example. Finally, we introduce our attempt to detecting the described problem.

3.1 Analyzed Frameworks

We searched different specifications and documentations for an evidence if the object-pool pattern may be applied in an implementation. We started with some core specifications, typically used in Java enterprise applications, such as the Enterprise JavaBean specification, the JavaServer Pages specification and the Java Servlet specification. Beyond that, we took a look at the Struts 1 framework, since it is used by our partners in the research project ASKS to implement their user interfaces.

Enterprise JavaBean An Enterprise JavaBean (EJB) is a managed component, running in an application container and follows a well-defined lifecycle that is steered by the application container. There are different commercial and non-commercial containers that implement one of the EJB specifications.

A special kind EJB is the stateless session bean, a component that is accessible for local and remote clients and implements business logic. The term "stateless" means that it does not hold a state to process a request sent by a client. Therefore, it is a candidate to be used within the object-pool pattern. The EJB specification in the version 2.1, as well as 3.0, states: "Since stateless session bean instances are typically pooled, the time of the clients invocation of the create method need not have any direct relationship to the containers invocation of the PostConstruct/ejbCreate method on the stateless session bean instance." [8, p. 8] and [9, p. 72]. This formulation does not prescribe the behavior of an application container, but gives a hint that the object-pooling mechanism may be used by application container implementations.

Depending on the version of the EJB specification, a stateless session bean must implement the interface `javax.ejb.SessionBean` or be annotated with `javax.ejb.Stateless`.

Java Servlet A Java Servlet is a managed entity that reacts on requests and produces some response that is returned to the origin of the request. An example is the `HttpServlet`-Interface that reacts on HTTP queries and an implementing class can generate any valid HTTP response. The Servlet API, as well as the lifecycle that a Servlet container has to provide, are defined in the Java Servlet specification. The specification mentions that the servlet container may choose to pool such objects [17, p. 7].

The base interface for all Java Servlets is `javax.servlet.Servlet`. To detect all Servlets, one has to find all classes that implement this interface.

JavaServer Pages JavaServer Pages (JSP) is a Java-based template language to easily generate dynamic web pages. A programmer can write a textual file which he enhances with some quoted Java code to insert dynamic data at runtime. A typical field of application of JSPs is the generation of HTML and XML documents, which are served to a web browser. According to the JavaServer Pages specification [23], the base interface for all JSPs is `javax.servlet.jsp.HttpJspPage` which is a Java Servlet. Therefore, JSPs are possibly pooled, too.

Struts 1 Apache Struts is a framework that implements the model-view-controller pattern [22] for JSPs. The framework uses traditional JSPs for the view part of the pattern and adds *Actions* to assume the controller role. An action is a Java class that inherits from `org.apache.struts.action.Action`.

The online documentation of Struts 1[1] says explicitly that one should not use instance variables within actions because of possible multi-threading issues that originate from the fact that each action is instantiated once and reused for all requests.

Remark The specifications mention that pooling of objects may be used which means that it is an implementation-defined behaviour of the frameworks or application containers. We inspected some open-source implementations of the specifications under investigation, such as the JBoss application container [21], the Glassfish application container, the Tomcat web server, and the Struts implementation and found out that all of them use object pooling.

A developer, who uses the aforementioned frameworks, does not explicitly use the pooling mechanism, instead he implements certain interfaces or uses some given annotations and the framework automatically pools instances of these objects. None of the specifications mentioned above provide some kind of reset method that is called when an object is returned to the pool and whose task is to clear the state of an instance.

3.2 Problem

As already mentioned, enterprise applications allow different users to connect simultaneously. Unique identifiers assigned to each user allow the container to distinguish the different users and provide an independent session for each of them. Ideally, each session would be handled in its own memory area to avoid private data to leak between sessions.

If the response time of a software system is an important non-functional requirement, software architects may decide to use the object-pool pattern since it can improve the speed of an application, by reusing class instances [25]. Normally, an object is temporarily bound to the client that is using the object, whereas we observed in the case of *Struts* that it is also used concurrently by different clients.

[1] http://struts.apache.org/1.x/userGuide/building_controller.html

The concept of object pooling contradicts to a strict separation of sessions since it explicitly shares object instances—which may contain sensitive information. This may lead to the aforementioned information flow between different sessions and therefore between different users, as long as the sensitive data is not removed when the instance is returned to the pool. This is not done by the frameworks we analyzed.

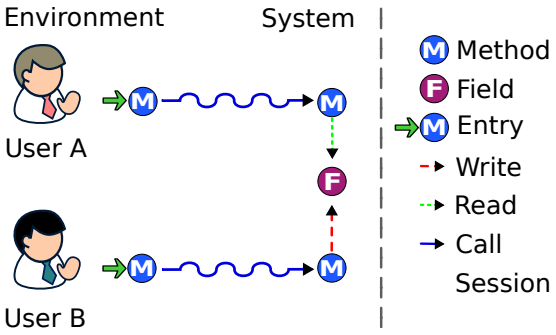

Fig. 1. Communication between sessions

The problem is depicted in Figure 1. Two different users are calling entry methods of a system, which call some internal methods to process the user request. Let us assume that the field on the right-hand side belongs to a pooled object. User B's request is processed first, an instance is fetched from the pool and a method is called that writes data to the field. The pooled instance now holds data of B and is returned to the object pool. The next request is issued by User A, who receives the same instance from the pool and calls a method that reads the data from the field and processes them. This constellation is a situation where sensitive information can leave the session (the framed area within Figure 1) of User A and flow into the session of User B or vice versa.

3.3 Example

A concrete example of an inter-session data flow problem is given in Listing 1.1, which shows a Struts action written in Java (see Section 3.1 for more information).

The action class `ExampleAction` has a private field `userEMail` of type `String` that can be accessed by the two methods `enterEMail` and `sendEMail`. These methods can be called by an HTTP request, depending on an HTTP parameter. The former method stores an e-mail address provided by the query into the field `userEMail` and does some additional operations, such as checking the provided value for correctness. The latter method uses the stored value from `userEMail` to send some private information to it.

The obvious problem in Listing 1.1 is the case in which User A sets her e-mail address and User B triggers the `send` method, which leads to the situation where data of User B might be sent to User A.

```
1  // imports are ommitted
2
3  public class ExampleAction extends Action {
4    private String userEMail;
5
6    public ActionForward enterEMail(ActionMapping mapping,
7                                    ActionForm form,
8                                    HttpServletRequest request,
9                                    HttpServletResponse response)
10                          throws IOException, ServletException {
11     EMailForm emailForm = (EMailForm)form;
12     this.userEMail = emailForm.getEMail();
13     ...
14     checkAndStoreEMailAddress(this.userEMail);
15   }
16
17   public ActionForward sendEMail(ActionMapping mapping,
18                                  ActionForm form,
19                                  HttpServletRequest request,
20                                  HttpServletResponse response)
21                          throws IOException, ServletException {
22     ...
23     sendEMail(this.userEMail);
24     ...
25   }
26 }
```

Listing 1.1. Struts example code

3.4 Detection Approach

To investigate whether the problem is relevant in real life, we implemented a detection algorithm to identify possible inter-session data flows using the Bauhaus tool-suite [20]. The basis for our analysis is a resource-flow graph (RFG), which contains typed nodes for every declared element and typed edges for relations between those elements. For the Java language, there are, for instance, nodes for classes, interfaces, methods, and fields, whereas edges represent call relations, inheritance, field access, or the used types. The call graph, for example, is a part of our resource-flow graph.

To construct an RFG, we use the application's bytecode, which gives us the possibility to analyze a software without having its source code. To analyze JSPs we use the JSP compiler Jasper from Apache Tomcat [28]. With the help of the JSP compiler a JSP file can be translated into normal Java source code, which can be processed by the default Java compiler. This way, we do not need extra support for JSP files. The RFG for an application is generated from the class files belonging to the system under investigation, as well as the depending libraries.

A simple approach for detecting such data flows would be to search for all pooled classes and report them if they contain a field. This way, one can identify all potential vulnerabilities, but it produces false positives since not every field leads to a flow of sensitive information. To reduce the rate of false positives, we implemented a detection algorithm that is a bit more sophisticated (see Figure 1).

The algorithm consists of three automatic successive steps. First, we identify all pooled instances within an application (lines 2 – 7). For this step, the method

isPooled uses the knowledge how to detect these objects which we have already described in Section 3.1. This step is similar to the naive approach. In the second step, we identify all methods that may be called by a user (lines 8 – 9)—in contrast to those methods that are called by the framework. These entries are the same ones that are used by injection-related analyses, for example, described by Livshits et al. [16]. Starting from these entry methods, we traverse the static call graph (call to function *getTransitivelyCalled* in line 10) to determine the methods that are reachable from the entry points. In the last step, we collect all fields, belonging to the pooled class that are accessed within the reachable methods (lines 11 – 20). Finally, a tuple is created, consisting of the accessing method and the access type (read, write), and is added to the list of results of the field that is accessed (lines 13, 18).

```
   Input: RFG Resource Flow Graph
   Output: Result A list of tuples, containing the method and the access type, for each field
 1 begin
 2     pooled ← ∅;
 3     foreach Class class ∈ RFG do
 4         if isPooled(class) then
 5             pooled ← pooled ∪ {class};
 6         end
 7     end
 8     foreach Method method ∈ RFG do
 9         if isEntry(method) then
10             foreach Method callee ∈ getTransitivelyCalled(method) do
11                 foreach Field field ∈ getReadFields(callee) do
12                     if getContainingClass(field) ∈ pooled then
13                         Result[field].append((callee, Read));
14                     end
15                 end
16                 foreach Field field ∈ getWrittenFields(callee) do
17                     if getContainingClass(field) ∈ pooled then
18                         Result[field].append((callee, Write));
19                     end
20                 end
21             end
22         end
23     end
24 end
```

Fig. 2. Detection algorithm for inter-session data flows

The results of our approach can be divided into four different classes, according to the fact whether they are read, written, or read and written.

read only The case that a field is just read from entries may be harmless, but not in all cases. If the field is a base data-type, such as *int*, *float* or *boolean* or it is an class instance that is not manipulated (for example, immutable instances), the field cannot lead to an inter-session data flow. In this case, the state of the program is not manipulated and therefore it is not possible for sensitive data to flow to another session. In the case that a class instance

is read, it is possible that a method of the instance is called that manipulates its state, which may lead to an inter-session data flow.

write only If a field is just written, there is no way for sensitive information to flow since no other session will read the data that may be stored in the object. Therefore, these findings are flagged as false positives.

read and write This group of results definitely can lead to data flows between sessions. The examples in Section 3.2 and Section 3.3 belong to this class.

no access If a field is not accessed at all, it poses no threat to the confidentiality of the data and therefore this is a false positive finding in the naive approach, too.

Our approach reports the groups "read only" and "read and write" as findings, in contrast to the naive approach described above. We are aware of the fact that the reported findings are not free of false positives and false negatives. This shortcoming will be discussed in Section 4.2.

4 Evaluation

For our evaluation, we considered three applications, two being open source and the other being proprietary software. The criterion for selecting these was whether they used one of the frameworks analyzed in Section 3.1.

4.1 Evaluated programs

The first open-source application we analyzed is the Java version of *enNode2* [2]. A running instance of *enNode2* is part of the Exchange Network, which has the aim to exchange environmental information between different States, Territories, Tribes, and the U.S. Environmental Protection Agency. The second application, called *GSS* [1], is a file storage service used for the Greek research and academic community.

The closed-source application is made available by one of our partners and is a successful commercial business application that helps companies to declare goods electronically for the import and export. The software is offered to customers on a software as a service basis, which makes the confidentiality of the user data an important and not easy to ensure requirement, due to the size of the existing source code (over 600k LoC and more than 1000 JSP files).

The different frameworks of interest that are used by the analyzed programs are listed in Table 1. Within the table, "EJBs" stands for stateless enterprise session beans, not for all type of existing Enterprise JavaBeans. One can see that *enNode2* is a web application that does not use Enterprise JavaBeans, whereas GSS does not employ the Struts framework.

Furthermore, Table 1 shows the size of the different systems, which was measured with `sloccount`[2]. One can see that the proprietary software is the largest application, by far. Additionally, the table shows the time that was

[2] http://www.dwheeler.com/sloccount/

Table 1. General information of the analyzed systems

Project	EJBs	JSPs	Servlets	Struts 1	LoC [k]	RFG [min]	Analysis [min]
enNode2		✓	✓	✓	85	2:27	0:43
GSS	✓	✓	✓		36	0:24	0:07
Commercial	✓	✓	✓	✓	614	4:42	1:51

necessary to construct the RFG for the program and the time our analysis took. The construction and analysis time shows that our approach is applicable for real-world systems.

4.2 Analysis Results

In the previous section, we showed which frameworks were used by which programs and gave a rough size estimation. Next, we highlight in Table 2 to which degree the classes that are implemented are pooled by a framework. 24% of all implemented classes in *enNode2* are pooled, and 68% of these have at least one field. In *GSS* just 8% of all classes are pooled, but each of these classes contains fields. Finally, *Commercial* has a pooled rate of 30% and a quite high rate of 84% of the pooled classes that contain fields. In addition, Table 2 shows that the classes that contain fields have a medium rate of fields.

Table 2. Frequency of pooled classes

Project	Num. of Classes	Num. of Pooled Classes	Pooled with Fields	Avg. Num. of Fields
enNode2	438	107 (24 %)	68 %	6.48
GSS	164	14 (8 %)	100 %	6.29
Commercial	4901	1485 (30 %)	84 %	8.72

In total, we found 473 fields of pooled classes within the implementation of *enNode2*, 88 fields in *GSS* and 10894 in *Commercial*. To determine whether the results may lead to inter-session data flows, we grouped the fields according to their usage as described in Section 3.4 and show the results in Table 3.

All fields that belong to the groups “write only” and “no access” are false positives and would have been reported by the simple approach in contrast to our approach. Therefore, our approach removes at least between 30% and 69% of the reported findings.

Table 3. Results grouped by access type

	enNode2	**GSS**	**Commercial**
read only	54.12 %	30.68 %	57.35 %
write only	0.42 %	0.00 %	4.56 %
read and write	4.23 %	0.00 %	11.92 %
no access	41.23 %	69.32 %	26.16 %

4.3 Discussion

With our approach, we can detect fields of pooled classes that may lead to inter-session data flow. Our approach has a lower false positive rate than the naive one since it identifies fields that cannot transfer data between sessions (see Section 3.4).

Nevertheless, the findings reported by our approach still contain false positives[3], since we just use control flow and field accesses for our detection. In order to improve the quality of our analysis, we are planning different enhancements that will be described in Section 5, such as a data-flow based approach. However, we were able to show with the help of our approach that the problem of inter-session data flow is relevant in the applications that we analyzed.

A possible reason for the quite high rate of unused fields may be the use of reflection, a typical source of false positives in static analysis. In some situations, it is not possible to determine which class is instantiated or which field is accessed. There is a work by Bodden et al. that uses run-time monitoring to track the aforementioned information and enhances the intermediate representation with the gathered details at analysis time to improve the results of static analysis [6].

To validate the results of our approach, we manually checked the assignment to the different groups of a part of our results and found that all checked findings were categorized correctly. During the checks, we noted that the classes generated by Jasper for the JSP files all contained generated fields that belonged to the read-only group. A manual inspection suggests that these fields do not transfer data between sessions and therefore are false positives that we cannot identify with our approach.

Beyond that, we inspected, in collaboration with the vendor, the reported problems of the commercial software. The vendor was able to identify several findings that have caused trouble in the past. In particular, customers complained that their data were messed up with data from other customers sporadically and the programmers were not able to identify the cause of the problem because they could not reproduce the symptoms, which the users reported. For some of our findings, we created automated reproducers that repeatedly called some functionality until the results indicated that the data do not belong to the current user. After removing the fields, we were unable to reproduce the erroneous behaviour which shows the effectiveness of our approach.

[3] Nota bene: The naive approach reports the same false positives.

5 Conclusion and Outlook

We described the problem of inter-session data flows that may arise from the wrong usage of the object-pool pattern in the context of different Java Enterprise frameworks. Furthermore, we developed a light-weight approach to detecting such flows that produces a lower rate of false positive than a naive approach and showed that this kind of problems appeared in open-source as well as in proprietary software. The size of our case study is relatively small and must be increased in our future work. Nevertheless, our approach helped us to identify and remove existing security holes in a commercial software.

In order to improve our approach, we are planning to implement several refinements. First, we plan to automatically detect the object-pool pattern, which makes the preliminary manual inspection of the framework documentation superfluous. Second, we are going to track the flow of user-related data through a system and find out whether the data are stored in memory locations that may be accessed from different sessions, in contrast to our current mere control-flow based approach. This task is similar to taint checking, employed to find injection vulnerabilities. Last but not least, we want to exclude the cases where checks are implemented which make sure the data cannot flow, such as locking and resetting areas that access the field.

References

1. File storage service with REST-like API, rich web GUI, webDAV. Online (Nov 2011), http://code.google.com/p/gss/
2. Open Source Exchange Network Node, supporting the National Environmental Exchange Network. Online (Nov 2011), http://code.google.com/p/en-node2/
3. Anderson, P., Zarins, M.: The codesurfer software understanding platform. In: Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on. pp. 147 – 148 (May 2005)
4. Ashcraft, K., Engler, D.: Using Programmer-Written Compiler Extensions to Catch Security Holes. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy. pp. 143–159. IEEE Computer Society, Washington, DC, USA (2002)
5. Bishop, M., Dilger, M.: Checking for Race Conditions in File Accesses. Computing Systems 9, 131–152 (1996)
6. Bodden, E., Lam, P., Hendren, L.: Clara: a framework for statically evaluating finite-state runtime monitors. In: 1st International Conference on Runtime Verification (RV). LNCS, vol. 6418, pp. 74–88. Springer (Nov 2010), http://www.bodden.de/pubs/blh10clara.pdf
7. Chess, B.: Improving Computer Security using Extended Static Checking. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy. pp. 160–173. IEEE Computer Society, Washington, DC, USA (2002)
8. DeMichiel, L.G.: Enterprise JavaBeans^{TM} Specification, Version 2.1. Sun Microsystems (2003)
9. DeMichiel, L.G., Keith, M.: JSR 220: Enterprise JavaBeans^{TM}, Version 3.0. Sun Microsystems (2006)
10. Feiman, J., MacDonald, N.: Magic quadrant for static application security testing. Tech. rep., Gartner, Inc. (2010)

11. Graf, J.: Speeding up context-, object- and field-sensitive sdg generation. In: Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on. pp. 105 –114 (2010)
12. Hammer, C.: Experiences with PDG-Based IFC. In: Massacci, F., Wallach, D., Zannone, N. (eds.) Engineering Secure Software and Systems, Lecture Notes in Computer Science, vol. 5965, pp. 44–60. Springer Berlin / Heidelberg (2010)
13. Hammer, C., Snelting, G.: Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. International Journal of Information Security 8(6), 399–422 (2009)
14. Kircher, M., Jai, P.: Pooling. In: Proceedings of the 2002 European Conference on Pattern Languages of Programs (2002)
15. Krinke, J.: Identifying similar code with program dependence graphs. In: Reverse Engineering, 2001. Proceedings. Eighth Working Conference on. pp. 301 –309 (2001)
16. Livshits, B., Lam, M.S.: Finding Security Vulnerabilities in Java Applications with Static Analysis. In: Proceedings of the 14th USENIX Security Symposium. pp. 271 – 286 (2005)
17. Mordani, R.: Java^{TM} Servlet Specification, Version 3.0 Rev a. Sun Microsystems (2010)
18. Nagy, C., Mancoridis, S.: Static Security Analysis Based on Input-Related Software Faults. In: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering. pp. 37–46. IEEE Computer Society, Washington, DC, USA (2009)
19. Oracle: Java EE at a Glance. Online (Nov 2011), http://www.oracle.com/technetwork/java/javaee
20. Raza, A., Vogel, G., Plödereder, E.: Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In: Proceedings of 11th Ada-Europe International Conference on Reliable Software Technologies. vol. 4006. Springer (2006)
21. Red Hat, Inc: Session EJB and MDB Configuration (2011), http://docs.jboss.org/ejb3/docs/reference/build/reference/en/html/session-bean-config.html
22. Reenskaug, T.: Models – Views – Controllers. Tech. rep., Xerox PARC (1979), http://heim.ifi.uio.no/ trygver/1979/mvc-2/1979-12-MVC.pdf
23. Roth, M., Pelegrí-Llopart, E.: JavaServer Pages^{TM} Specification, Version 2.0. Sun Microsystems (2003)
24. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons Ltd (2006)
25. Souza, F., Arteiro, R., Rosa, N., Maciel, P.: Performance Models for the Instance Pooling Mechanism of the JBoss Application Server. In: Performance, Computing and Communications Conference , 2008. IPCCC 2008. IEEE International. pp. 135–143 (2008)
26. SpringSource: SpringSource.org. Online (Nov 2011), http://www.springsource.org
27. The Apache Software Foundation: Apache Struts. Online (Nov 2011), http://struts.apache.org
28. The Apache Software Foundation: Apache Tomcat. Online (Nov 2011), http://tomcat.apache.org/
29. Wassermann, G., Su, Z.: Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 32–41. PLDI '07, ACM, New York, NY, USA (2007)