A Secure OSGi Environment for Untrusted Web Applications

Timo Aho
Tampere University of Technology
P.O.Box 553
FI-33101 Tampere
Finland
timo.aho@tut.fi

Johannes Koskinen
Tampere University of Technology
P.O.Box 553
FI-33101 Tampere
Finland
johannes.koskinen@tut.fi

Antti Nieminen
Tampere University of Technology
P.O.Box 553
FI-33101 Tampere
Finland
antti.h.nieminen@tut.fi

ABSTRACT

For some time it has been a growing trend to move applications from the desktop to the web and especially to cloud environment. Very often the web application solutions are based on the Java language. In this case, the OSGi specification is arguably the number one choice for running multiple applications on a single Java virtual machine. Unfortunately, OSGi does not solve all the security vulnerabilities that emerge in such an environment. For instance, computer resource usage is only marginally controlled. In this paper, we discuss the security of the OSGi environment. In particular, we introduce a solution to running untrusted OSGi applications. In our case, controlling the permissions of the applications is fairly simple. A more challenging task is to manage the computer resource usage. We present a moderately straightforward solution that still grants a reasonable level of security. Unlike other similar OSGi resource managers and monitors, our solution does not need any modifications to the web applications or OSGi components. Moreover, we distinguish each web session of an application while competing methods only monitor complete applications as single entities.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*protection mechanisms*

General Terms

Measurement, Performance

Keywords

Java, Cloud Computing, OSGi, Profiling, Web Applications

1. INTRODUCTION

One of the main trends in current development in the computing world is the move toward ubiquity. This has been made possible by the platform independent development technology — especially web and cloud techniques. The significance and amount of web applications is growing rapidly; more and more traditional desktop applications are moving to the web. This trend has multiple notable benefits. Web applications are available geographically everywhere and independent of the underlying platform. The web, and especially the usage of the cloud [1, 2], makes thin application clients possible by moving functionality to the server side. Altogether, the web provides new inexpensive ways to publish and maintain applications.

A remarkable part of web applications is based on the Java programming language. Especially Java *servlets* [8] are a technique which is widely used for web applications. Some of the key advantages of using the servlets are the huge number of tools and frameworks available to help you develop web applications with it. An important factor in these techniques is a *web container*, which is the component of a web server that interacts with the servlets. A web container is responsible for managing the lifecycle of servlets and mapping a URL to a particular servlet. When the HTTP request from the browser is received by the web server, it is forwarded to the web container. The container maps this request to a particular servlet and calls its method in order to process the request.

In order to share the same network address and to conserve the resources of the host server, the servlets share the same Java virtual machine (JVM) instance. Each HTTP request is serviced in its own separate thread. In this way, the different requests can be serviced simultaneously. However, because of the shared JVM, the resources of the JVM are shared. In practice, this means that the servlets share the memory available, the CPU-time and access rights of the JVM. If some of these resources are somehow limited by the operating system or the runtime environment, the limitations will apply to all of the servlets, and thus all of the web applications, executed in that JVM. Moreover, one cannot use operating system level (i.e. JVM level) information to monitor the resource usage as there is no way to tell which web application is using the resources. Sharing the resources also means that if some of the web application exceeds the limits set to JVM, all the web applications are closed as the JVM instance is ended.

Even if the web applications were distributed in the different Java virtual machines so that each web application is executed by separated virtual machines in cloud environment, closing one of the JVMs would still affect all the users using the application. In other words, all the sessions, i.e. instances of that web application are closed.

The Java language lacks many important supportive features to run multiple web applications on a single Java virtual machine. To patch this up, a widely adopted *OSGi*[1] specification [19] has been published. OSGi introduces a dynamic component model which allows, e.g., updating, installing and removing components on-the-fly in a single Java virtual machine [5, Section 15.1]. The components, also called OSGi *bundles*, can communicate with each other in a dynamic fashion. There are multiple implementations for the standard: e.g., open source Apache Felix[2], Makewave Knopflerfish[3], and Eclipse Equinox[4] as well as commercial ones.

OSGi is compatible with Java servlets. In fact, OSGi provides a core bundle that implements a web container to run Java servlets so that web applications can be deployed as OSGi bundles. Thus, the web applications do not need to care about the OSGi specific issues.

Despite its many merits, OSGi unfortunately does not solve all the security problems attached to multiple public untrusted web application hosting. Some of the vulnerabilities are implied by Java principles [3, 6], some by the OSGi features [14, 15]. For example, the usage of computer resources like memory, CPU time, network traffic is only marginally controlled in plain Java [11].

As the direct usage of servlets is relatively rare nowadays, we use a Java framework called Vaadin [4] to test the ideas presented in this paper. Vaadin is an open source framework for developing servlet based Rich Internet Applications (RIA). Vaadin framework is also readily compatible with OSGi. Because the framework is configured as an OSGi bundle it is possible for other bundles to use it directly. In this case, the applications use the Vaadin framework bundle like a dynamic library resulting in very lightweight applications. The typical size of a Vaadin application bundle file is less than 100 kilobytes, so the application starts up in an instant as the Vaadin framework code is already in memory and ready for use. As OSGi provides means to load and register servlets dynamically [5, Section 15.1], we use a special OSGi registration bundle for Vaadin applications.

In this paper, we discuss the security of running web applications on the OSGi platform. As a concrete case, we use a public cloud-based web hosting service. The users can deploy their own web applications directly to the hosting service. After that, the web applications are dynamically loaded by the OSGi server environment. This kind of service will clearly require some kind of security and resource management so that malfunction in one web application cannot jeopardize the functionality of the rest of the web applications or the hosting system. Moreover, if only one session of the web application takes too many resources, only that session should be ended, not the whole application. Analogously, if one of the Google Docs sessions takes too much memory, it is not an option to close the whole service from all of the Google Docs users.

The open hosting service should take account at least the following issues:

1. How can we restrict file access of the web applications so that the applications cannot read or write files owned by others? In addition, the system configuration files should not be available for the hosted web applications.
2. How can we set separate limits for resource usage to each web application or even to each web application instance (session)? These limits should be also updateable.
3. How can we find out the web application session that has exceeded its limits and close it?
4. How can we monitor the resource usages of each web application?

To address these issues, we introduce solutions and an example implementation of such security decisions. The paper is organized as follows. The security issues and our solutions for them are split in two categories. In Section 2, we go through the permission control of untrusted and trusted OSGi bundles. In Section 3, we introduce the more complicated matter: computer resource monitoring and controlling. Finally, we present some concluding remarks and discussion in Section 4.

2. OSGI SECURITY AND PERMISSIONS

In publicly available services, we have to prepare for the malicious and defective behavior of client web applications. This can either be a result of user actions with a malfunctioning or vulnerable web application or even a hostile application loaded on our hosting server.

By default, OSGi controls the client applications only slightly. All the running code has, e.g., a permission to call the `System.exit` method, which terminates the execution of the whole Java virtual machine. An exhaustive list of the vulnerabilities of the OSGi environment has been presented in [14, 15]. Fortunately, the OSGi Release 4[5] Specification [19] gives at least a partial solution by defining an optional OSGi Security layer. In our implementation, we use the publicly available Apache Felix implementation which includes the layer [5].

The OSGi Security model is based on the Java 2 Specification [10]. The basic idea is that code has to be authenticated

[1] OSGi originally stood for Open Services Gateway initiative framework. The longer form is at present seldom used because the specification has moved beyond the original focus.
[2] Available at `http://felix.apache.org`.
[3] `http://www.knopflerfish.org`
[4] `http://www.eclipse.org/equinox`
[5] As of 2012, also OSGi Release 5 has been published. Nevertheless, OSGi Certification only covers releases until version 4.2. Moreover, implementations also seem to support at most Release 4.

before it is given *permission* to execute method calls outside its own domain. The model supports directly authenticating by file location and by the signer of the JAR files. Moreover, custom authentication types can be created by uniting conditions. We refer to literature [5, Section 14.7] for more details. In addition to the original Java permissions, the OSGi security layer introduces some OSGi specific ones. However, they follow the same principles.

The Java permission model is based on a class called `Permission`. All the specific permissions are derived from it. The class has a method called `implies` which takes another permission as an argument. It indicates whether the argument permission is implied by the first permission. This way the class takes care of both defining and checking permissions. The OSGi framework further simplifies the Java permission model by treating every bundle as a single separate permission entity.

Now, when a potentially problematic method is called, the virtual machine checks that the whole call stack has permission to execute it. If this is not the case, the operation is denied. There is also a technique called *privileged calls* to truncate the call stack passing. For further details, see [5, 10, 19]. The same guidelines are followed regardless of the type of service applied for. The called method may be offered internally by Java, by external libraries, by the OSGi framework or by other OSGi bundles.

All the bundles are granted all permissions by default. All the entities can also define global permissions and so first one to do so makes the decision. Thus, by using OSGi `StartLevel` package [19, Section 8.6] we have to take care that our security bundle is started before external applications. In Felix, this can be done with an implementation specific configuration file. In our application, all the untrusted external web application bundles are installed in a directory separate from our framework. Hence, for bundles in the directory, we can deny all the permissions not necessary for basic starting and running. When using an openly available web service to publish web applications, we cannot trust the client applications at all.

Nevertheless, what are the necessary permissions? In our case, the Vaadin applications primarily need permission to import and register Vaadin services. The OSGi framework takes care of other basic privileges like starting up and making operations on a private bundle cache on disk. In this kind of public environment, the external bundles should have very limited permissions. For instance, they should not interact with each other, start new bundles or use disk space in paths outside their private area. A normal permission file could be like in Listing 1.

If the web application code is offered by an unknown author, the OSGi bundle package should be wrapped up and validated by our own framework. In practice, this means that static analysis methods are applied to bundles to be hosted. Furthermore, if the compilation of the web application to bytecode is also carried out by our service, it is possible to restrict the libraries and method calls available for the web application. In this way, hostile actions by, e.g., bundle manifestation files are at least very challenging to

```
ALLOW {
[org.osgi.service.condpermadmin.BundleLocationCondition
 "${FELIX_FRAMEWORK}/bundle/*"]
( java.security.AllPermission "" "")
} "All permissions to internal application JARs"

ALLOW {
[org.osgi.service.condpermadmin.BundleLocationCondition
 "${FELIX_FRAMEWORK}/load/*"]
( org.osgi.framework.PackagePermission "com.vaadin" "import")
( org.osgi.framework.PackagePermission "com.vaadin.*" "import")
( org.osgi.framework.ServicePermission "com.vaadin.Application"
  "register")
} "Allow the basic permissions to external applications"

DENY {
[org.osgi.service.condpermadmin.BundleLocationCondition
 "${FELIX_FRAMEWORK}/load/*"]
( java.security.AllPermission "*" "*")
} "Deny all other permissions for external applications"
```

Listing 1: An example permission file

execute. Also the bundle size can be controlled beforehand. These techniques prevent a significant amount of the vulnerabilities listed in [15].

Nevertheless, the above-mentioned solution is still inadequate in some respects. First of all, the presented security solution does not control all the resource usage of bundles. Thus, a bundle can still use uncontrollable amount of memory, CPU time, disk space in private area and execute unrestricted amount of method calls. This problem is partially addressed in our resource manager introduced in the next section.

Some more minor vulnerabilities exist. For example, we rely on Apache Felix implementation specific behavior on the starting order of the bundles. This behavior is not explicitly defined by the OSGi specification. If the starting order is changed, an external bundle could be used to manage the rights.

In addition, we come across some vulnerabilities introduced by Java platform, especially the ones mentioned in the justification of *Java specification request* (JSR) 121 for application isolation [6]: At least some parts of the Java virtual machine state can be, often unintentionally, shared between applications. With the use of the shared state it is possible to affect other applications and even the virtual machine itself in unwanted ways. There are several attempts [3, 17], most notably Multi-tasking virtual machine [18], to implement JSR-121. Unfortunately, there is no up-to-date implementation on Java 2 Standard or Enterprise editions. Fortunately, both the use of OSGi and strict permissions substantially limit the conceivable vulnerability in our case.

3. MONITORING RESOURCE CONSUMPTION

Even though we can limit what the applications are allowed to do with Java permission management, there are still areas that cannot be covered by the permissions. One of the most important areas is the resource consumption of the web applications. It is important that one application or application instance cannot "steal" all the memory or CPU time of the Java virtual machine. This is even more criti-

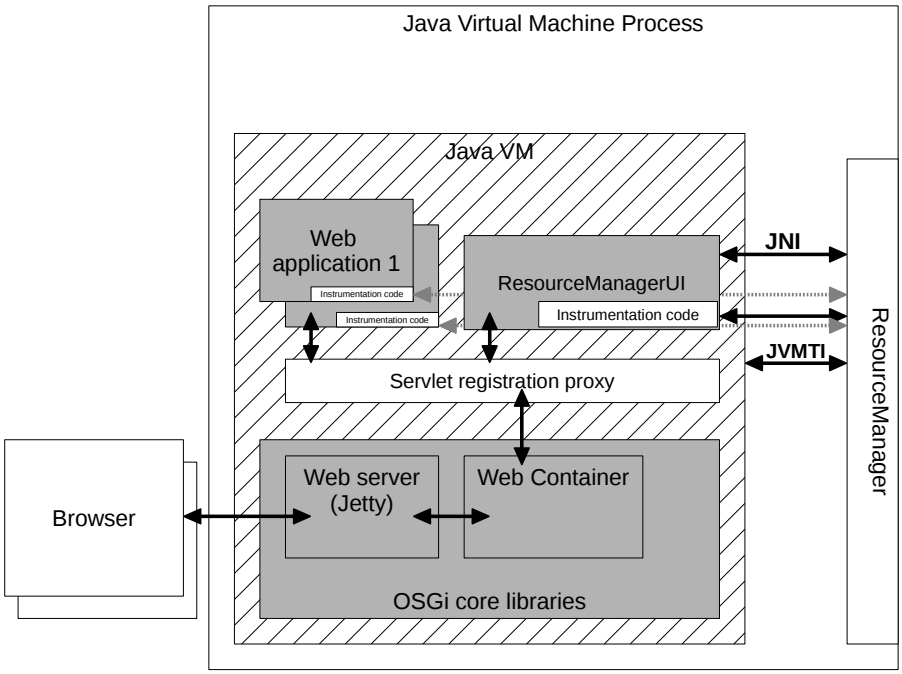

Figure 1: Overview of Resource Manager.

cal in cloud computing environments where customers are charged on the used computation time.

For monitoring resource consumption, Resource Consumption Management API (JSR-284) [7] has been proposed. Unfortunately, we are not aware of any working implementation of the specification available for the OSGi environment. For this reason, we have implemented our own resource monitoring component called Resource Manager. Figure 1 illustrates the components of Resource Manager and their relationships.

Resource Manager is focused on monitoring resource consumption of the web applications. Unlike other similar OSGi resource managers and monitors (such as [3,9,11,16]) it does not need any modifications to the Java virtual machine, existing web applications, or OSGi components. Moreover, our purpose is to monitor and collect information on the web application sessions, not the bundles themselves.

Resource Manager will not try to restrict the usage of external resources like files as they are already handled by the permissions. Instead, the focus is on the internal resources of the virtual machine. Such resources are the number of threads, used CPU time (per thread and per application), and allocated memory. These resources are selected as the monitoring of them can be done without any knowledge of the application and will not cause too much overhead. In addition, these resources are shared by all the web applications inside the Java virtual machine. There are still other resources, namely disk space and network, whose usage should be monitored by using some other means.

3.1 Implementation of Monitoring

In practice, Resource Manager is a simple profiling tool, which is augmented with per application limits. When one of the application limits is reached, Resource Manager tries to interrupt or stop the misbehaving application. Moreover, the usage information can be presented to the administrator and logged to be used later for analysis purposes. For example, it might be reasonable that the applications requiring very much CPU time are run on separate servers.

Resource Manager is based on the *Java Virtual Machine Tool Interface* (JVMTI) [13]. JVMTI provides a way to inspect the state of applications running in the Java virtual machine. Resource Manager is hooked to a Java virtual machine so that when the virtual machine starts a new thread, Resource Manager is called by the JVMTI. Similarly, Resource Manager is called when threads are stopped. Memory allocations can be easily covered by instrumenting the code to call Resource Manager via *Java Native Interface* (JNI) [12] whenever a new object is allocated. The instrumentation is carried out when the class file loaded, so no preliminary changes to the code are required. To track the release of the memory, one can use JVMTI callbacks. Another alternative is to iterate through heap memory and calculate memory blocks that are in actual use. We have used the former, faster method, even though it may temporarily give inaccurate memory consumption figures as the memory blocks are released only by the garbage collector.

Monitoring CPU time is a little more tricky, as the usage must be polled on regular intervals. This is normally done by having a separate thread which periodically collects the

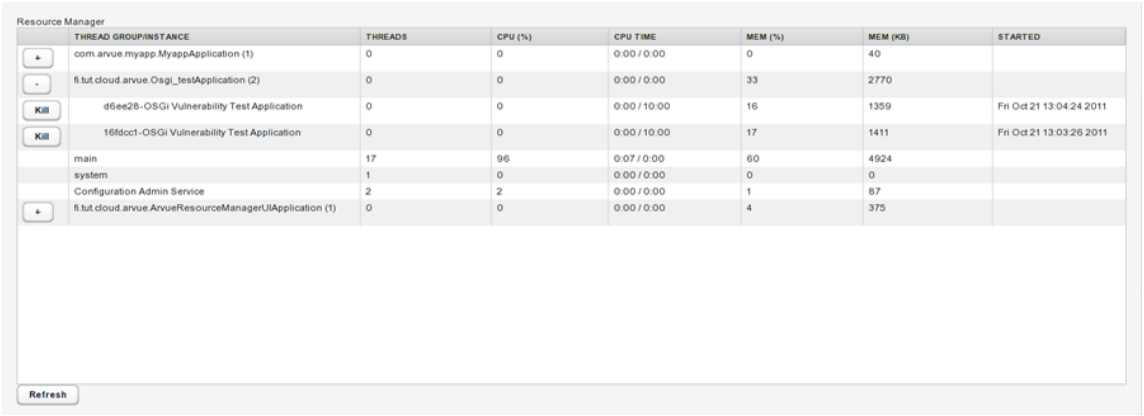

Figure 2: Monitoring Web Application for Resource Manager.

used CPU time on all the threads that are active on that time. This approach works very well on background threads, as they are active quite long time. For short-lived threads, it might be possible that the thread is started just after the collection process and is ended before the next round is started. In this case, the collector thread does not even notice such a thread. For our purposes, this is not such a big problem as we are interested in excessive CPU consumption, and thus we do not have to worry about Resource Manager occasionally missing a short-lived thread.

The client applications are not aware of the resource monitoring. Neither does Resource Manager have any detailed information on the applications. Hence the basic JVMTI calls are not enough to find out which application has allocated a certain object. Resource Manager has only following information about the memory allocations: currently using thread, a partial call stack, and the amount of the allocation. The situation is even worse with new threads as they do not contain any information on the parent threads which originally applied for them. For this reason, we use the only easily available method to distinguish applications, namely *Java thread groups*. Thread groups are, as the name describes, a mechanism to collect a set of threads. By default, we assign a thread to the group which includes its parent. At least one group always exists because the Java runtime system creates a thread group named `main`. By having a unique thread group for each application, the amount of allocated resources can be calculated by using the thread group as criterion. The idea can be extended so that each opened application session has its own thread group.

Every time a service from a web application servlet is required, a web container calls the service method of the servlet. Each request is serviced in its own separate thread or, in other words, each call is made by separate thread. Unfortunately, it is not possible to change the thread group of a running thread. However, we can handle this method in a proxy which starts a new thread in web application instance/session specific thread group and starts running the original service in the new thread. It is relatively easy to have separate thread groups for each web application, even without any modifications to existing code if a registration proxy bundle is used. Moreover, now all resource allocations are made in the context of the new thread group and thus in the context of the session or application instance. A pseudo code for the proxy is presented below:

```
service(request,response):
  -- find the correct session based on the request
  Let applicationSession=getApplicationSession(request)
  Let threadGroup=getOrCreateThreadGroup(applicationSession)
  if threadGroup is destroyed, end session and return
  Let t=new thread in threadGroup
  -- call the original service method
  t.start(application.service(request, response))
  t.join() -- wait for thread to end
  if t was interrupted:
    throw a new exception("Resource limits exceed")
  if session has ended:
    destroy threadGroup
```

With Vaadin applications, we can use a dynamic servlet registration bundle with a service call proxy to add monitoring aspect to all Vaadin based web applications. With other frameworks or web applications made without any supporting framework, the proxy code should be included to a proper place to be called before `javax.servlet.service`. It should be noted that this proxy code will work regardless of Resource Manager component being loaded or not. However, it would be more or less useless without the profiling component.

3.2 User Interface for the Administrator

To provide feedback for the administrator, an interface is required for an administrator component to get the resource consumption information. An example, *ResourceManagerUI*, in shown in Figure 2). As Resource Manager is a native library residing outside of the virtual machine, JNI interface must be used. We can fetch the information about a single thread group (e.g., for the specific web application) or about all the collected applications. If the thread groups are named according to the application names and the session ids, the administrator gets the information in human readable form even though Resource Manager knows nothing about the actual applications. Remember that Resource Manager only

notices single threads and thread groups but does not know about their conceptual link with the applications.

A natural user interface for information provided by Resource Manager is a regular web application itself. It gets the information using the JNI interface and shows the results on the screen. If the user wants to kill a selected application session, it is carried out by destroying the corresponding thread group. The proxy code presented before notices the destroyed group and ends the session. Destroyed groups are removed from the database of Resource Manager component by garbage collector thread, which compares existing thread groups with the collected information and removes all information for the non-existing groups.

3.3 Monitoring Overhead

Monitoring and profiling will always cause some overhead. In our case, the overhead consists of three different components: starting a new thread in the registration proxy, monitoring threads with JVMTI callbacks, and monitoring memory usage via instrumented code. Starting a new thread has previously been quite a complex operation and finding or creating a correct thread group for the thread will add the overhead even further. However, creating a new thread with current versions of Java virtual machines causes negligible overhead. In addition, the thread is started only after an HTTP request, so their frequency is reasonable and the overhead will remain at acceptable levels. Based on our tests, the overhead of starting a new thread is less than 0.4 ms per request in a relative slow machine with Java 1.6.

Nevertheless, two remaining overhead sources are more problematic. JNI and JVMTI calls can be time consuming as they cross the virtual machine boundary even though they still stay inside the virtual machine process. However again, starting a new thread is a moderately light operation, and even with native calls the thread monitoring overhead (less than 1 ms including the creation of the new thread) is almost always overshadowed by the actual thread execution time (typically hundreds of milliseconds).

Unfortunately, monitoring memory usage may cause problems. If the application allocates thousands of objects for each HTTP request, the allocations may impose significant overhead on program execution. In the worst case, where the test application did nothing but allocated resources, the execution time of the application was multiplied. For such applications, it might be reasonable to leave instrumentation phase out, thus disabling memory usage monitoring. Alternatively, one can regard such an application as the hostile one and decline to start the application.

4. CONCLUSIONS

In this paper, we discussed the security features of OSGi based web applications on a publicly available web application hosting service. To test our ideas, we used applications based on Vaadin framework [4]. The idea can easily be generalized to all web applications in OSGi environment. In fact, in other environments only the functionality of servlet registration proxy needs to be implemented in some other way as discussed in Section 3.1.

In the introduction, we discussed the security issues to be solved. The first issue covers the restriction of file access of the untrusted web applications. In our application, we are able to solve permission based vulnerabilities of this kind moderately simply with the OSGI permissions.

We encounter a more challenging task when trying to solve the rest of the issues. In order to address the resource related security issues, we monitor the applications for an excessive usage of computer resources like memory and CPU time. If an application exceeds its limits, the session of the application is automatically closed to protect the other applications that are executed on the same Java virtual machine.

We ended up with a tool set, which contains three different components: a servlet registration bundle with a service call proxy to separate HTTP requests for different web applications and their sessions, Resource Manager for actual profiling and monitoring tasks, and a user interface to show the current profiling information.

We have noticed that this kind of monitoring and profiling will cause some overhead. Part of the overhead comes from the service proxy, the rest from the code instrumentation. Based on the tests, the separation based on the thread groups has negligible overhead. The monitoring code for memory allocation causes more overhead as the web application usually allocates several short-lived resource objects during the service handling. However, the overhead will not be a major problem with regular web applications. In addition, Resource Manager provides an option to skip the memory monitoring.

In summary, in this paper we presented a simple security solution for hosting OSGi based web applications. It grants at least moderate level security for the system. However, it is worth noting that the security level implied by our approach is not quite as high as the one introduced by isolation based systems, e.g., [3]. However, unlike other similar OSGi resource managers and monitors like [11] our solution does not need any modifications to the web applications or OSGi components. Moreover, our solution also treats every web session of each application as a monitoring target which is often more useful for a security point of view than solely the summary of the application usage.

5. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] Y. Chen, V. Paxson, and R. H. Katz. What´s new about cloud computing security? Technical Report UCB/EECS-2010-5, University of California, Berkeley, CA, 2010.

[3] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *International Conference on Dependable Systems and Networks (DSN 2009)*, Los Alamitos, CA, 2009. IEEE Computer Society.

[4] M. Grönroos. *Book of Vaadin*. Vaadin Ltd., Turku, Finland, 7th edition, 2013.

[5] R. Hall, K. Pauls, S. McCulloch, and D. Savage. *OSGi in action: Creating modular applications in Java*. Manning Publications, Greenwich, CT, 2010.

[6] Java Community Process. Java specification request 121: Application isolation API specification, 2006. Version 2.7, final.

[7] Java Community Process. Java specification request 284: Resource consumption management API, 2009. Version 2.6, final.

[8] Java Community Process. Java specification request 315: Java servlet 3.0 specification, 2009. Version 3.0, final.

[9] H. Lin, C. You, M. Zhou, and H. Mei. Proxy centric approach for component resource monitoring on OSGi platform. *Journal of Frontiers of Computer Science and Technology*, 5(1):23–31, 2011. In Chinese with English abstract.

[10] S. Microsystems. Java 2 security architecture, 2002. Version 1.2.

[11] T. Miettinen. Resource monitoring and visualization of OSGi-based software components. Technical Report 685, VTT, Espoo, Finland, 2008.

[12] Oracle. Java native interface specification, 2006. Version 6.0.

[13] Oracle. JVM tool interface, 2006. Version 1.2.1.

[14] P. Parrend. *Software Security Models for Service-Oriented Programming (SOP) Platforms*. PhD thesis, Institut National des Sciences Appliquées de Lyon, Lyon, France, 2008.

[15] P. Parrend and S. Frénot. Java components vulnerabilities: An experimental classification targeted at the OSGi platform. Technical Report 6231, Institut National de Recherche en Informatique et en Automatique, Le Chesnay Cedex, France, 2007.

[16] R. Schwammberger. Performance isolation for component systems. Master's thesis, Swiss Federal Institute of Technology Zurich, Zurich, Germany, 2009.

[17] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: The Squawk Java virtual machine. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE 06)*, pages 78–88, New York, NY, 2006. ACM.

[18] S. Soman, L. Daynès, and C. Krintz. Task-aware garbage collection in a multi-tasking virtual machine. In *Proceedings of the 5th International Symposium on Memory Management (ISMM 06)*, pages 64–73, New York, NY, 2006. ACM.

[19] The OSGi Alliance. OSGi service platform: Core specification, 2009. Release 4, version 4.2.