Intra-Engine Service Security for Grids Based on WSRF

Matthew Smith, Thomas Friese, Bernd Freisleben
Dept. of Mathematics and Computer Science,
University of Marburg
Hans-Meerwein-Str, 35032 Marburg, Germany
Email: {matthew, friese, freisleb}@informatik.uni-marburg.de

Abstract

In typical on demand grid computing scenarios, services from different organisations can potentially run in the same web service engine on a single grid node, making intra-engine service security vital for any production system. In this paper, a solution to the problem of intra-engine inter-service security for ad hoc grid environments based on WSRF is presented. To ensure that only authorized access to grid services is possible from within other services' code, a dynamic group enabled sandboxing approach within Apache Axis is proposed to protect dynamically deployed grid services. It relies on the features provided by a hot deployment service developed for ad hoc grids. A prototypical implementation of the hot deployment service and the intra-engine service security approach based on the Globus Toolkit 4 (GT4) is used to demonstrate the feasibility of our approach.

1. Introduction

The service-oriented architecture (SOA) approach and the corresponding web service standards such as WSDL [1] and SOAP [2] are currently adopted in various fields of distributed application development (e.g. enterprise application integration, web application development, inter-organizational workflow collaboration). The Open Grid Services Architecture (OGSA) [3, 4] incorporates the web service paradigm in the field of grid computing as an approach towards defining the *service-oriented grid*. The OGSA effort to add stateful interaction to the web service environment has also been recognized by other web service users not focused on grid computing. As a result, the specifications of the Web Service Resource Framework (WSRF) [5] have emerged.

The WSRF introduces the notion of a *web service resource* (WS-Resource) that is formed by the combination of a *resource document* and a corresponding *web service*. It is the purpose of the resource document to capture state information for a WS-Resource while the corresponding web service remains stateless. In this way, a multitude of WS-Resources can be created using one stateless web service implementation which captures the state of execution in multiple resource documents. The WSRF further defines web service interfaces to inspect and alter the information contained within the resource document and to receive and subscribe to notifications on property changes.

The service-oriented grid paradigm offers the potential to provide a fine grained virtualization of the available resources to significantly increase the versatility of a grid. For instance, all idle workstations of a company may be combined in a dynamically formed *ad hoc grid* to handle computational peak loads, thus extending the maximum computational power of the company without further expense. This brings the service-oriented grid a step closer to fulfilling IBM's vision of *on demand* computing [6].

In previous work [7] we introduced the concept of an ad hoc grid as a possible environment for on demand computing. We also presented a solution to the problem of *hot service deployment*, which is one of the basic requirements for such a flexible grid computing environment [8]. Our work on hot service deployment raised some of the security issues and attack scenarios that are being addressed in this paper, and in fact our approach to provide intra-engine service security is based on the features offered by the solution developed for hot service deployment.

Current work on security issues in WSRF focuses on the enforcement of access restrictions and protection of message exchanges in transit. Implementations of the WSRF specifications do not address issues concerning intra-engine service security, since providing such mechanisms is not enforced or encouraged. Therefore, it is possible for various service implementations to directly access each other through simple method calls, bypassing the service security mechanisms established for access control. The underlying system model assumes a single grid node to be a unified compartment containing only trusted users and resources. To some extent, this is a dangerous assumption even in a conventional grid environment where two

different users share the same resource. In an ad hoc grid environment where the sharing of a grid node between multiple parties is the desired usage pattern, the establishment of secure and trusted compartments is a neccessity for user acceptance.

In this paper, we address potential attack scenarios between grid services running in the same web service engine and present our approach to provide intra-engine service security. A dynamic group enabled sandboxing approach within the Apache Axis web service engine [9] is proposed to protect dynamically deployed grid services. The approach relies on the features provided by our hot deployment service previously developed for OGSI based ad hoc grids, and we address the changes required by the transition from OGSI to the WSRF. A prototypical implementation of the hot deployment service and the intra-engine service security approach based on the Globus Toolkit 4 (GT4) [10] is used to demonstrate the feasibility of our approach. Our solution is not limited to the ad hoc grid scenario; confinement of services and separation of instances is also beneficial in a scenario where two separate users share a single grid node in a traditional grid. Ultimately, an extension of the WSRF standard by specifications governing the discussed service security issues of grid services based on WSRF would be the most preferable outcome.

The paper is organized as follows. In section 2, the notion of a service-oriented ad hoc grid environment and scenarios for potential intra-engine service attacks are discussed. Section 3 presents the extended functionality provided by our hot deployment service in a GT4 environment. In section 4, we introduce our solution to the intra-engine security issues. Section 5 discusses related work. Section 6 concludes the paper and outlines areas for future research.

2. Background and Problem Statement

2.1. The Ad Hoc Grid Environment

An ad hoc grid is a spontaneous organization of cooperating heterogenous nodes into a logical community without a fixed infrastructure and with only minimal administrative requirements. The ad hoc grid idea goes beyond a static grid infrastructure to encompass frequent dynamic additions to the grid. This includes workstations within organizations as well as scattered personal computers similar to the idea of many popular distributed computing projects [11]. The goal of the ad hoc grid is to provide computing resources on demand to every member of the ad hoc grid. Unlike traditional grid systems, the number of non-dedicated nodes is much higher, demanding non-intrusive operation of the ad hoc grid middleware. Our definition of an ad hoc grid is discussed in more detail in [7].

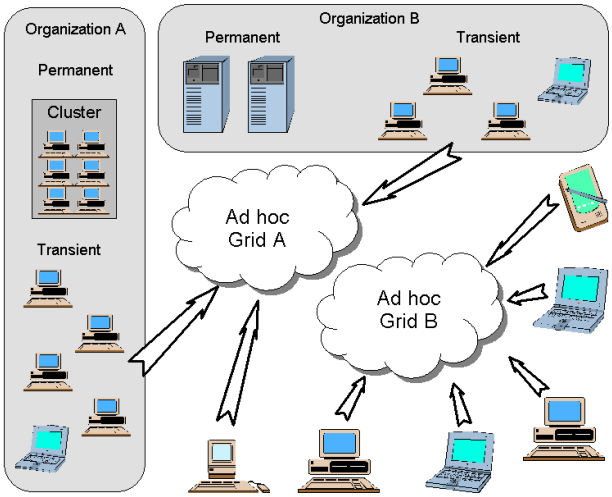

Figure 1. Two dynamic ad hoc grid environments.

Figure 1 shows how two separate ad hoc grids are composed spanning two organizations respectively scattered nodes on the Internet. Both grid communities form a virtual organization using the existing Internet infrastructure. While ad hoc grid A encompasses transient nodes it also includes a dedicated high performance computing base. In contrast, ad hoc grid B is made up solely of transient individual nodes. While ad hoc grid A bears a greater resemblance to traditional grid systems, ad hoc grid B illustrates the shift to a personal grid system, built without the resources of a large organization.

The OGSA specification considers virtualization at multiple levels and allows for the creation of such an ad hoc grid with only minor extensions. Looking at the ad hoc grid infrastructure from an application perspective, the following considerations are particularly important:

Vital to the on demand invocation of services on various nodes in the grid system is the availability of those services. For a large scale ad hoc grid, the time consumption for manual deployment is prohibitive and management is difficult due to the fluctuating availability of the nodes; it also runs contrary to the on demand philosophy. Even with the availability of advanced grid programming toolkits, deployment of services has been identified as a critical issue [12]. In a dynamically changing environment, deployment is even more critical as there is no single deployment cycle that reaches all machines, instead services need to be deployed and instantiated on demand on machines as they become available. The step of service deployment becomes part of an ad hoc grid application instead of being handled by a system administrator as a precondition to the use of a service. Instead of only providing predefined services, the computational nodes of the bare grid become a resource by themselves. When a node becomes available that meets the

```
package de.fb12.grid.services;
class A{
    static protected Data getData()
    { ... }
    static protected void setData(Data data)
    { ... }
    public void doSomething() throws RemoteException{
        //do something good
}}
```

Figure 2. Target Service Main Class

```
package de.fb12.grid.services;
public class EvilService{
   doDataAttack(){
       stolenData=A.getData();
       A.setData(evilData);
}}
```

Figure 3. The Data Attack Service

requirements for the deployment of a service, the application can autonomously carry out the deployment and use the newly available machine for its application flow.

Security is a major aspect in an ad hoc grid since there is always the possibility that a node introduces malicious code. On demand computing with dynamic service deployment creates several new security aspects that must be dealt with beyond the standard security requirements existing in previous grid systems. Like in traditional systems, installing a service requires security certificates allowing the operation. In traditional Grids, services usually can only be installed by a small number of people and trust may be assumed between all parties. In a large ad hoc grid system offering on demand service deployment, it is possible that users unknown to each other operate on the same node. Thus intra-engine inter-service security must be offered, since fair play is not guaranteed any more.

2.2. Service Security Threats

In this section, the security threads are presented in more detail. Since the separate grid services running on one grid node are all hosted by Axis in GT4, they run within the same JVM and the classes are loaded by the same class loader. As a consequence, interaction between the classes is possible, thus offering malicious code the possibility to harm running services. To illustrate the problem, we introduce a target service which we will then attack. The service code is listed in 2. The main class A lies in the package de.fb12.grid.services. It has two static methods for service internal use to access and set some data. It also has a public service method which is called by the clients wanting to use the service. The get and set methods are declared as protected methods, which restricts access to these methods to classes declared in the same package.

The static get and set methods are the origin of the first intra-engine service security issue and enable an *intra-engine service data attack*. In general, an intra-engine service data attack is made possible if the service targeted for attack uses singletons, or any other method to access internal objects which does not require specific object references, like static methods. It is then possible to introduce a service which can modify these objects simply by using the same package as the target service and calling the methods on those objects. Using this attack, the internal state of an object belonging to a different service can be modified. Listing 3 shows how data from the target service in figure 2 is accessed and then replaced by a different data object.

Since the get and set methods are declared as protected, the attacking class was placed in the same package as the target class. In a standard computing environment, we could configure a security manager to restrict access to the package de.fb12.grid.services and thus prevent the malicious class entering our package and accessing our target service. This is done by setting package.definition=de.fb12.grid.services and package.access=de.fb12.grid.services in the java.security properties file, effectively blocking creation of new classes in the protected package and access to the package from unauthorized classes. These security properties must be set before starting the system. In an on demand grid environment, this is not feasible since new services in custom packages will have to be inserted during runtime and thus could not be protected. Furthermore, legitimate access to these existing services or legitimate deployment of new services in the same package should be possible. The standard approach in Java to allow this is to set specific security permissions (shown in figure 4) for the code base trying to access a package or create a new class in that package. In the case of GT4, which uses Tomcat as the WebApplication host, these settings are stored in catalina_home/conf/catalina.policy and are loaded during startup. Again, this is not a usable solution for on demand grid computing since code bases not registered during startup might need legitimate access to certain packages. To prevent illegal access we require a sandboxing system which allows us to protect classes from intra-engine service data attacks while at the same time allowing dynamically added services to access our classes if they have sufficient security clearance and the certificates to prove it.

A different form of attack is the *intra-engine service code attack*. If the target service is loaded after the attacking services or loads classes on demand (the standard procedure in Java), it is possible to introduce foreign code into the service by preempting the load procedure. If, for example, the target service has a Class A which loads Class B at

```
grant codeBase "file:\${wsrf.home}/WEB-INF/lib/MyService
my-service.jar" {
  permission java.lang.RuntimePermission
  "defineClassInPackage.de.fb12.grid.services"};
```

Figure 4. Excerpt from the catalina.policy file

a certain time during its operation, a malicious service can use that as an entry point for an attack. Figure 5 illustrates the attack. The attacking service defines a Class B* with the same fully qualified name and with the same method signature as Class B and then loads that class (5.1). If this is done prior to the loading of B in the target service, the malicious code has been successfully introduced into the system (5.2). When A tries to load B (5.3), the ClassLoader will see it has already loaded a class with the fully qualified name of B (namely B*) and returns the class from its cache (5.4). As a consequence, A now executes B* instead of B (5.5). This attack can even be used to replace the service A completely if A is deployed to a node where the malicious service is already running. Listings 6 and 7 show how this is done. Listing 6 shows the malicious replacement A* of the target service A. It has the same method signature as the legitimate service A and thus can function as its replacement but executes the attack code instead when, for instance, the method doSomething is called. If the target service is deployed onto a grid node where the malicious service A* is already loaded, the Axis class loader will simply return A* when A is supposed to be loaded, since A* has the same fully quallified name as A. Listing 7 shows how the attacking service loads the A* class.

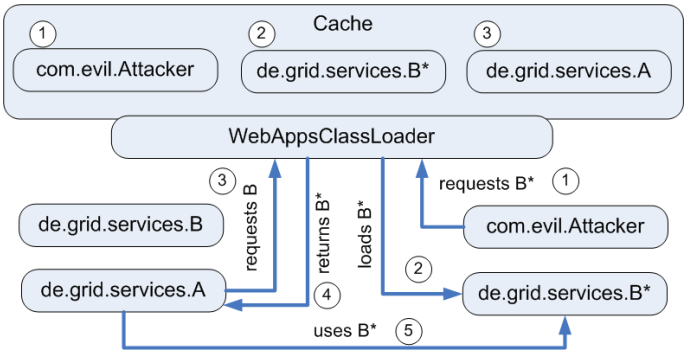

Figure 5. Intra-Engine Service Code Attack

If service code is protected by placing the computation into a separate class which can only be instantiated by classes of the same package, as shown in listing 8, the attacking service can either be placed in the same package or it can avoid instantiating the class altogether and simply load the class explicitly, as shown in 9.

The solution to the above security problems requires

```
package de.fb12.grid.services;
class A{
    static Data getData()
    { ... }
    static void setData(Data data)
    { ... }
    private void doSomething() throws RemoteException{
        //do something malicious
}}
```

Figure 6. Malicious Replacement of the Target Service Main Class

```
package com.evil;
import de.fb12.grid.services.B;
public class EvilService{
    private doCodeAttack(){
        A a = new A();
}}
```

Figure 7. The Code Attack Service

```
package de.fb12.grid.services;
class B{
  protected B()
    { ... }
   private void doMore(){
       //do something good
}}
```

Figure 8. Target Service Helper Class B

```
package com.evil;
public class EvilService{
  private doCodeAttack(){
    try{
      EvilService.class.getClassLoader().
        loadClass("de.fb12.grid.services.B");
    } catch (ClassNotFoundException e){
      e.printStackTrace();
}}}
```

Figure 9. The Code Attack Service 2

that we are able to deploy services into separate sandboxes which protect the services from illegal access. The next section presents the hot deployment service which is later utilized by our security infrastructure to do that.

3. Hot Service Deployment in GT4 Ad Hoc Grids

A WS-Resource is the collection of a web service and a resource document holding the state information of the resource. Access to the service part of a resource in GT4 is provided by the Axis JavaProvider. A newly created WS-Resource is uniquely identified by a WS-Addressing Endpoint Reference (WS-ER) that can later on be used to interact with the resource and distinguish between different instances for different users. Every Globus service is registered with the GT4 web application by making an appropriate web service deployment descriptor (WSDD) entry available to the web application container hosting the Globus web application.

Vital to the creation of new resource instances is the availability of the accompanying web service and the schema definition of the associated resource document. Hot service deployment addresses the need to dynamically and non-intrusively provide the service implementation and schema definitions in a running grid container. We have previously addressed this issue in an GT3 environment [8]. In contrast to GT3 where a custom Servlet extending the AxisServlet was used to invoke the service, GT4 passes the request to the standard AxisServer, thus creating a much cleaner system. GT4 now only needs to deal with the grid specific requirements.

The deployment of a service currently requires a grid service archive (GAR file) containing the needed classes, schema files and deployment descriptors that make up a service. Users of GT4 are supplied with Ant tasks that handle the distribution of the contents of this GAR file into the local standalone GT4 environment. The Ant tasks extract and copy the jar files containing the class files of the service into the local web application directory. Schema files are copied into the schema repository. The current deployment strategy of GT4 requires the restart of the entire WSRF web application, thereby killing every other grid service currently running. Furthermore, direct access to the machine running the GT4 application is required because the Ant tasks perform all copy operations locally.

Neither the first nor the second property of the deployment mechanism are feasible for an ad hoc environment with a frequently changing collection of nodes. In this environment, an application has to make sure - through dynamic service deployment - that the required service is present on every node it wishes to incorporate into its application flow.

To enable this, we modified the Axis web service engine utilized by GT4 to allow dynamic loading and unloading of grid services. Our *hot deployment service* (HDS) provides applications with the capability to remotely *deploy, undeploy* and *redeploy* services onto a running node. The operations have the following semantics:

Deploy adds a service to the set of available services on the grid node. The service is identified by its service name. The operation will not deploy the service if there is a service with the same name already present on the node.

Undeploy removes a service from the node, based on its service name. Running service instances already created are unaffected by the operation.

Redeploy is the chaining of undeploy and deploy. Running service instances are not changed by the redeploy operation, subsequent requests to create new instances will, however, use the newly supplied implementation of the service.

In our current implementation, access to the HDS is restricted by using the security mechanisms offered by GT4.

The basic steps the HDS needs to perform to deploy a service are:

- Register the service description with the AXIS/WSRF request handlers.
- Make the schema files available to the WSRF environment.
- Make the service class files available to the class loader.

Currently, the need to load additional classes and dynamically replace them was not anticipated or governed by the WSRF specification or the GT4 implementation. To enable this functionality, a novel class loading mechanism is introduced into the realization of the HDS, as described in the following.

Grid services in GT4 are separated into three classes: The service resource class, a service home class and the service implementation itself. The service home class is used to load resources attached to a service and the service classes themselves. We provide the class HotResourceHomeImpl as our implementation of the ResourceHome interface in order to leverage our own class loading mechanism into GT4. The ResourceHome is responsible for creating the ClassLoader hierarchy. It distinguishes different instances of the ClassLoaders by acquiring the service context from the AXIS engine inside the GT4 web application. It also registers all ClassLoaders created by it at a central DisposableClassLoaderManager and the Axis ClassUtils ClassLoader cache, so they can be accessed later during undeployment. The code snippet in figure 10 shows the

```
public void setResourceClass(String clazz)
    throws ClassNotFoundException {
String serviceName =
  AxisEngine.getCurrentMessageContext()
  .getTargetService();
String basePath =
  ContainerConfig.getConfig().
  getInternalWebRoot();
String relPath   basePath+servicePath+libPath;
ClassLoader cl =
  JarClassLoaderManager.createLoader(
  serviceName, relPath)
resourceClass = cl.loadClass(clazz);
}
```

Figure 10. The setResourceClass operation in HotResourceHomeImpl

main operation of our HotResourceHomeImpl. First, the service name is extracted from the current message context. Then, the path where the jar files of the service are stored is generated based on the container configuration and an arbitrary path extension. In our case we chose basePath/WEB-INF/lib/serviceName/. Based on that, we create a JarClassLoader capable of loading all classes contained in all jar files in that directory. The JarClassLoaderManager also informs the Axis ClassUtils that it is now responsible for this service.

This is a non-intrusive way to introduce our own class loading mechanism into GT4, since the ResourceHome implementation can be specified for each individual service. A service wishing to be hot deployable merely must use the HotResourceHomeImpl instead of the standard ResourceHomeImpl. This is the only change required to make a service hot deployable and reloadable. Hot deployable and standard services can be run side by side by using the different ResourceHome implementations. Figure 11 shows the relationship of the ResourceHome implementations and class loaders.

The process of loading a service class is as follows. When a Service is first requested, the org.globus.wsrf.jndi.BasicBeanFactory loads our HotResourceHomeImpl class in the standard Axis WebAppsClassLoader. The HotResourceHomeImpl is responsible for creating the disposable ClassLoaders wich will later load the service classes and the attached resources. When the setResourceClass method is called by the BasicBeanFactory, the CurrentMessageContext from the Axis engine is parsed to discover on behalf of which service the method is being called, thus allowing us to create one and only one ClassLoader for each service. Our JarClassLoaderManager and the modified Axis ClassUtils are informed of the service to ClassLoader mapping. Once the ResourceHome is in place, the BasicBeanFactory informs the home object which class is the main service class. As mentioned above, the HotResourceHomeImpl attaches a disposable ClassLoader to the service. Class and ClassLoader are then used by the org.globus.axis.providers.RPCProvider to instantiate the actual service object. The HotResourceHomeImpl makes sure that the class is loaded in the proper ClassLoader. Now everything is in place and the service can be accessed via the JavaProvider.

Removal of a service is mainly concerned with removing the service entries from the in-memory registries. Once the service information has been removed, no new service instances can be created. Running instances of a service previously created are untouched by this process. To deploy a new version of a service, no explicit unloading of the old service classes is required, since the new version of the service will be created using a new ClassLoader. If in addition to the service information the service instances are to be removed, the central manager used by the JarClassLoaderManager can be used to access the ClassLoaders of the separate services to free the resources and unload the classes. Only then can the jar files be deleted, since otherwise active services might try to lazy load classes after the containing jar files have already been removed.

On top of the deployment service and the ClassLoading structure described above, our solution to intra-engine service security is realized, as described in the next section.

4. An Approach to Intra-Engine Service Security

The attacks described in 2.2 lead us to propose the following intra-engine service security requirements: A service must be able to be deployed into a private sandbox if it does not want its classes to be accesed by other services. Services wishing to form a group in which classes can be shared require a secure grouping mechanism which allows all services within the group to share classes but services outside of the group are denied access. Both mechanisms must function in an on demand fashion, i.e., normal operation of the grid node must not be disrupted. Services already running on the system must be unaffected by the introduction of new services and new security groups.

In GT4, the intra-engine service attacks described in section 2.2 are made possible by the fact that GT4 loads all grid services within the same class loader. The basic idea of our solution to this problem is to use the introduced ClassLoader hierarchy to enable the dynamic loading and reloading of classes to also provide intra-engine service security. In its most basic form, each grid service is loaded within its own ClassLoader functioning as a sandbox and as such its classes and resources are private and can not be accessed by any other service. This ensures that services using singletons can not be hijacked by malicious services, and foreign code can not be inserted into the program flow.

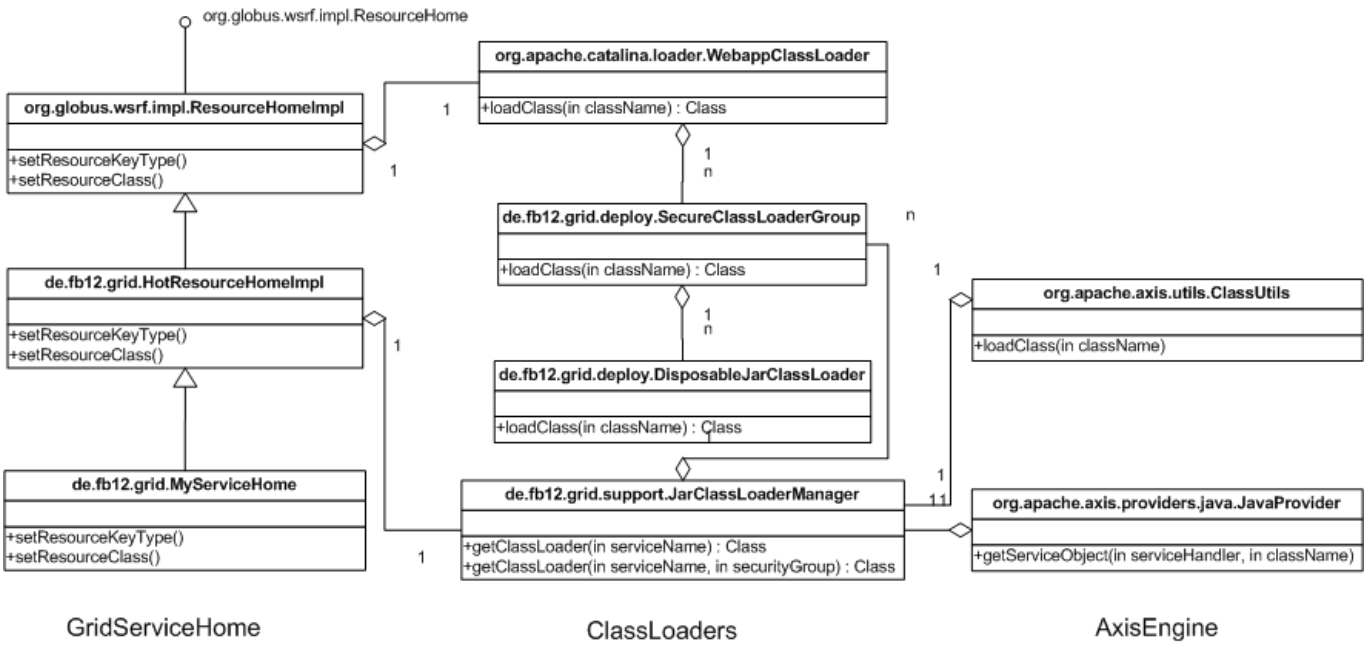

Figure 11. Relationship of the ResourceHome implementations and class loaders.

4.1. Service Sandboxing in Axis

To enable the required secure loading process, the ClassUtils and JavaProvider classes provided by Axis needed to be modified. The code snippet in figure 12 shows the necessary changes to the loading mechanism in the Axis ClassUtils. To ensure that service classes are only loaded by our sandboxed class loader, the current message context is checked as to whether the current loader request was triggered by a service or by the container itself. If it was triggered by a service, it checks whether it is a service which is registered with our ClassLoaderMananger and should be protected. If that is the case, the class is loaded using the appropriate service ClassLoader and Axis is prevented from loading the classes in its WebAppsClassLoader and thus breaking our sandbox. Otherwise, the Axis class loading is unmodifed, allowing all normal operations to proceed unhindered. The second place where Axis could try and load the service classes into its own WebAppsClassLoader is the JavaProvider. Listing 13 shows the modification to the loading process needed to protect the service classes. Similar to above, we check whether the service is registered with our framework and if that is the case we preempt the Axis loading mechanism and use our own ClassLoaders.

Through these modifications to Axis, inter-service communication is now confined to using web service calls and thus ensures that proper authentication between services must be observed. In many cases, this approach suffices to protect the service being deployed while still allowing unhindered operations within the service.

```
static Class loadClass(String _className)
 throws ClassNotFoundException {
 final String className = _className;
 // Get the class within a doPrivleged block
 Object ret = AccessController.doPrivileged(
   new PrivilegedAction() {
   public Object run() {
    try {
    MessageContext mc =
      AxisEngine.getCurrentMessageContext();
    if(mc!=null)
    {
     String serviceName =mc.getTargetService();
     if(JarClassLoaderManager.
       isRegistered(serviceName))
     {
      ClassLoader classLoader =
        JarClassLoaderManager.
        getJarClassLoader(serviceName);
      return Class.forName(className, true,
        classLoader);
     }
    }
    ClassLoader classLoader =
      getClassLoader(className);
    return Class.forName(className, true,
      classLoader);
    } catch (ClassNotFoundException cnfe)
    { ... }

   ...
}
```

Figure 12. The modified loadClass operation in the Axis ClassUtils

```
protected Class getServiceClass
  (String clsName,
   SOAPService service,
   MessageContext msgContext) throws AxisFault {
  if(JarClassLoaderManager.
      isRegistered(service.name)){
    ClassLoader cl = JarClassLoaderManager.
      getClassLoader(service.name)
    return cl.loadClass(clsName)
  } else {
    standard Axis behaviour
}}
```

Figure 13. The modified getServiceClass operation in the Axis JavaProvider

4.2. Secure Sandbox Groups

If it is necessary that two services be able to communicate directly using class references to create a composed web service application, they must group their ClassLoaders together using a SecureGroupClassLoader provided as part of our intra-engine service security infrastructure. A service specifies which group it wants to join either by passing the groupId to the Hot Deployment Service or by setting the parameter `<parameter name="group" value="groupId"/>` in the server-deploy WSDD of the service. The SecureGroupClassLoader responsible for the group is a parent ClassLoader to all service ClassLoaders in that group. It enables inter-service communication in two ways: First, separate communication classes are placed in the SecureGroupClassLoader which can be accessed by all child ClassLoaders. This is the preferred way as defined by Java to allow classes in sister ClassLoaders to communicate. For instance, the interface class of an object to be used by classes in two sister ClassLoaders is placed with the parent so it can be accessed by both children. The implementing classes are placed in both child ClassLoaders and object references can be passed between ClassLoaders as long as only the interface defined in the parent ClassLoader is used. This is the traditional way to allow code-based interaction between services but it requires that the communication classes are placed in the parent ClassLoader. The disadvantage of this approach is that if the communication classes need to be replaced, all child ClassLoaders must be discarded because the SecureGroupClassLoader must be replaced. So, even if only two services use the communication classes, all services must be undeployed to update the communication classes. To avoid this problem, the SecureGroupClassLoader is capable of emulating a flat namespace for its child ClassLoaders while still allowing HotDeployment and HotUndeployment of component parts of the composed web service application.

When a service ClassLoader joins the SecureGroupClassLoader, the SecureGroupClassLoader checks which classes the service ClassLoader is capable of loading and stores that information internally. If a different service within the same group tries to load one of those classes, its own ClassLoader will not be able to find the class and thus asks its parent, the SecureGroupClassLoader. The SecureGroupClassLoader then checks whether one of the other service ClassLoaders can load the requested Class and passes the request on to that ClassLoader before passing the request on to its parent ClassLoader, the WebAppsClassLoader. This, of course, only works if each Class is only defined once within all ClassLoaders in the same group. If different versions of one and the same class can be accessed from the same ClassLoader, TargetInvocation and ClassCastExceptions will be the result. However, this ClassLoading mechanism was designed to allow tightly coupled web services to be composed into a web application, so it is very unlikely and undesirable that the same class will be defined in two different places, since the idea of tight integration was to be able to reuse the classes of the other services. In the case of such class duplication, the less tightly coupled composition via service calls is the preferred way of linking different web services, and the grouping function should not be used.

4.3. Undeployment of Grouped Services

Undeployment of services is more complex if the service to be undeployed is in a group, since classes from services loaded in different ClassLoaders can have references to each other. To prevent these classes from being undeployed and crashing the system when one of the other services tries to access undeployed classes, the SecureGroupClassLoader stores the information which service ClassLoaders have interacted with each other and denies undeployment requests to these services unless all service ClassLoaders which are coupled to it are undeployed at the same time. Services loaded in the same group but which have not accessed classes from the Services to be undeployed remain unaffected by this process. This is a clear benefit compared to the standard approach of placing the communication classes in the parent ClassLoader.

As an example, figure 14 shows four grid services which are joined into a group by one SecureGroupClassLoader. Service A defines classes U and V where U uses W which is defined by Service B. Service C defines classes X and Y and Service D defines class Z. Class Y uses Z and Z uses X. That means, Service B can not be undeployed while A is alive, and Services C and D can only be undeployed together.

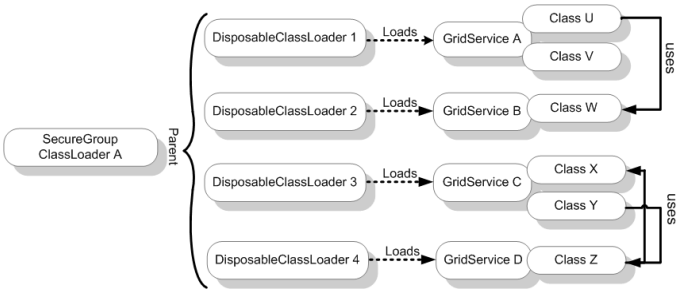

Figure 14. ClassLoader Group Interaction

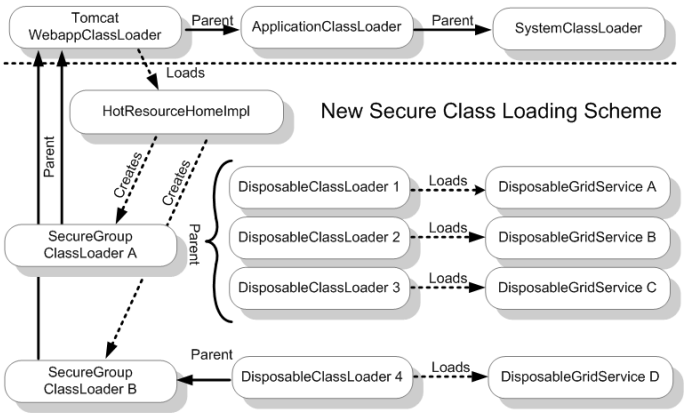

Figure 15. Hierarchy of the ClassLoader instances.

4.4. Group Access

To be able to securely group different service ClassLoaders together, access control to the grouping function is required. In our current implementation, when a group is created it has one owner who gets an asymmetric key pair to enable access control to the group. The private key is used by the group owner to sign Grid Service Archives (GARs) which are to be admitted to the group. The public key is used to identify the group and to check whether the GARs submitted for deployment are permitted to join the group in question. When the deploy method in the HotDeploymentService is called, the HotDeploymentService checks wether the GAR submitted for deployment was signed by the private key using the public key for that group. If the GAR was signed correctly, the deployment process is allowed and the service ClassLoader is added to the SecureGroupClassLoader of that group; if not, the deployment process is aborted and no changes to the grid environment are made.

Figure 15 shows a snapshot of the complete ClassLoader hierarchy in the system, after four different grid services have been instantiated in two separate security groups. Services A through C are deployed in the same group and thus can access each others' Class definitions. Service D is deployed in its own group and thus is protected from direct code access by any of the other services deployed on this node.

The above solution to inter-service security shows one possible way of protecting services from attacks within the same web service engine on which the service is running. Since with the progressive adoption of grid technologies in the scientific and business communities, intra-engine inter-service security will become more relevant as more users will share grid nodes. It would be best if the WSRF specifications deal with this topic. We propose that the requirements posed at the beginning of this section be formulated in a platform independent way, which nonetheless binds WSRF implementations to enforce intra-engine inter-service security on all platforms. The specification should then be integrated in the WSRF specifications family.

5. Related Work

The Open Grid Services Architecture (OGSA) has been accepted as the foundation for service-oriented grid computing. While OGSA describes the higher level architectural aspects of service-oriented grid computing, the Web Service Resource Framework (WSRF) is a fine grained specification of the infrastructure required to implement the OGSA model. Several implementations of WSRF are being developed concurrently, including: Globus Toolkit 4 (GT4) [10] and WSRF.NET [13].

The Globus Toolkit 4 (GT4) offers mechanisms to create service instances on a grid node via the Axis Java Provider. GT4 does not offer a flexible mechanism for hot service deployment (i.e., the deployment and removal of a grid service without restarting the GT4 engine inside the web application container). Instead, a service is deployed using an Apache Ant [14] task. The files making up a grid service are copied into the GT4 directory structure. To effect the changes, the application container must restart the GT4 application, causing a restart of every service running in its context. Intra-engine service security is not dealt with by GT4.

The architecture of the WSRF.NET grid uses the Microsoft Internet Information Server (IIS) as its hosting container. Dynamic deployment of services and intra-engine service security are not addressed by WSRF.NET, and the deployment process is similar to a Tomcat based GT4 distribution. It requires the alteration of the container's configuration file, copying of service descriptions as well as service assemblies (service DLLs) and the restart of the IIS.

The attacks described in section 2.2 and our security solution presented above demonstrate that intra-engine inter-service security is an issue which should be dealt with by

the WSRF community. The current WSRF specifications [5] do not deal with intra-engine inter-service security at all, leaving an intra-engine inter-service security framework completely to the hands of the different WSRF implementations. It is highly likely that different implementations would deal with the issue in different ways, creating a non-homogenous security environment. That would significantly weaken the WSRF environment in its entirety, since WSRF alone would not represent a secure grid environment. Clearly, it should be possible to deploy a service to a WSRF compliant grid node, sure in the knowledge that intra-engine inter-service security is a guaranteed feature of the environment.

6. Conclusions

In this paper, it was demonstrated how easy it is to hijack a service running on the same web service engine as an attacking service. A sandboxing security mechanism to ensure intra-engine inter-service security has been proposed. It allows services to be deployed on demand into a running webservice engine of a grid node, either joining an existing security group using the right credentials or creating a separate group to ensure inter-service security for all services. No direct code access between services is possible unless the service was signed using the appropriate group authorization, in which case only members of that group can access the classes belonging to the service. The intra-engine inter-service attacks presented in this paper can not be executed on a grid node using this security mechanism. The combination of the standard security model of GT4 and the proposed ClassLoader-based sandboxing and grouping of services offers the necessary security for on demand service-oriented grid computing.

Future work will consist of formulating the intra-engine service security requirements in a WSRF-* specification consistent way and possibly submitting these requirements to the WSRF standards group. To examine the usability and robustness of our system, we will proceed to test the introduced security mechanism in production environments. Finally, we intend to consider possible additonal intra-engine threats to grid services and develop solutions to them, to ensure that widespread adoption of WSRF based Grids is not hindered by platform security deficiencies.

Acknowledgement

This work is partially funded by Siemens AG, Corporate Technology, Munich, Germany.

References

[1] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," 2001, http://www.w3.org/TR/wsdl.

[2] The World Wide Web Consortium, "Simple Object Access Protocol (SOAP)," 2003, http://www.w3.org/TR/soap/.

[3] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," in *Open Grid Service Infrastructure WG, Global Grid Forum*, 2002.

[4] I. Foster, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, H. Kishimoto, F. Maciel, A. Savvy, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. von Reich, "The Open Grid Services Architecture, Version 1.0," 2004, https://forge.gridforum.org/projects/ogsa-wg/document/draft-ggf-ogsa-spec/en/.

[5] OASIS, "Web Services Resource Framework," 2004, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.

[6] IBM, "E-Business on Demand: The Race is on," 2003, iBM Whitepaper, http://www-5.ibm.com/e-business/de/literature/.

[7] M. Smith, T. Friese, and B. Freisleben, "Towards a Service-Oriented Ad Hoc Grid," in *Proceeedings of the 3rd International Symposium on Parallel and Distributed Computing, Cork, Ireland*, 2004, pp. 201–209.

[8] T. Friese, M. Smith, and B. Freisleben, "Hot service deployment in an ad hoc grid environment," in *Proceedings of the 2nd Int. Conference on Service Oriented Computing, New York, USA*. ACM Press, 2004, pp. 75–83.

[9] Apache Software Foundation, "Apache Web Services Project," 2004, http://ws.apache.org/axis/.

[10] T. G. Project, "The Globus Toolkit 4.0 (GT Version 3.9.3)," 2004, http://www-unix.globus.org/toolkit/downloads/development/.

[11] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "SETI@home - Massively distributed computing for SETI," *Computing in Science and Engineering*, vol. 3, no. 1, p. p. 79, 2001.

[12] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski, "Grid Web Services and Application Factories," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. John Wiley & Sons Inc., December 2002.

[13] D. Byrne, A. Hume, and M. Jackson, "Grid Services and Microsoft .NET," in *Proc. of UK e-Science All Hands Meeting*, 2003, pp. 129–136.

[14] The Apache Software Foundation, "The Apache Ant Project," 2004, http://ant.apache.org.