February 2007

Native Code Security for Java Grid Services

Thomas Friese
Siemens AG, th.friese@siemens.com

Matthew Smith
University of Marburg, matthew@informatik.uni-marburg.de

Bernd Freisleben
University of Marburg, freisleb@informatik.uni-marburg.de

In: Oberweis, Andreas, u.a. (Hg.) 2007. *eOrganisation: Service-, Prozess-, Market-Engineering*; 8. Internationale Tagung Wirtschaftsinformatik 2007. Karlsruhe: Universitätsverlag Karlsruhe

ISBN: 978-3-86644-094-4 (Band 1)
ISBN: 978-3-86644-095-1 (Band 2)
ISBN: 978-3-86644-093-7 (set)

Native Code Security for Java Grid Services

Thomas Friese

Siemens AG, Corporate Technology, Information and Communications,
Otto-Hahn-Ring 6, D-81539 Munich, Germany
E-Mail: th.friese@siemens.com

Matthew Smith, Bernd Freisleben

Dept. of Mathematics and Computer Science,
University of Marburg, D-35032 Marburg, Germany
E-Mail: {matthew, freisleb}@informatik.uni-marburg.de

Abstract

In modern on demand grid computing scenarios, services from different organisations will potentially run on the same web service engine of a grid node. Secure isolation of data and code of different service instances is a vital requirement in such an environment, since mutual trust cannot be assumed between all involved parties. For Java based Grid applications the Java virtual machine offers sandboxing facilities, however the common occurrence of native code (e.g. C/C++, Fortran) in business and scientific Grid applications leads to a number of security issues which are not handled by the basic Java security mechanisms. In this paper, we analyze the threat scenarios that emanate from native code in a service-oriented Grid scenario. A novel security architecture is presented, which enables a fine grained confinement of native components of Grid applications into a secure environment for protecting the hosting system as well as other service instances. Although our work focuses on Grid services, it is also relevant for any hosting scenario in which multiple web services using native code components are deployed in the same service container.

1 Introduction

The service-oriented architecture (SOA) and especially the web service paradigm have been adopted in various fields of distributed systems development. With the advent of the service-oriented Grid computing paradigm (i.e. the Web Service Resource Framework - WSRF), the

web service standards have also been applied in the field of scientific and high performance computing, extending their adoption beyond the fields of enterprise application integration or general business computing.

Java based web service containers have been widely accepted for the provisioning of web services. Services become small self-contained units in this scenario. They encapsulate the business logic represented in the web service, the services use the infrastructure services provided by the service container in which they are deployed.

In previous work [1] the concept of an ad hoc Grid as a possible environment for on demand computing was introduced. A solution to the problem of hot service deployment, which is one of the basic requirements for such a flexible Grid computing environment [2] was presented. The fact that services can be deployed to a number of nodes and migrate freely between them during an applications lifecycle leads to a number of security issues that need to be dealt with. The security domain is divided in two parts related to pure Java implementations of Grid services and non-Java native code.

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. Our work on hot service deployment raised some of the security issues and attack scenarios that we have addressed and solved for pure Java services [3].

Apart from the Java based implementation of web service containers, the .NET environment offers the functionality to provide web services. For service-oriented Grid computing, the dominant middleware frameworks (e.g. Globus Toolkit 4 and Unicore GS) have chosen Java based implementations. Thus, in this paper we focus on a Java based solution.

Many business and scientific applications in Grid computing are based on legacy code bases which cannot be ported to Java for cost and efficiency reasons. The existing legacy solutions are usually wrapped with Java Grid service implementations to make them available to the service-oriented Grid environment. This creates a major security problem since the native code cannot be constrained by the standard Java security facilites.

In this paper, we analyze the threat model induced to the service-oriented Grid by native legacy code and present a solution for imposing fine grained security enforcement on such native

service components. Our solution allows to host different Grid services containing native code in the same hosting environment while ensuring that integrity of the hosting environment cannot be damaged by individual services; the services are kept in 'compartments', such that a service cannot attack another service. The proposed security mechanisms increase the resilience of the service hosting environment against both malicious attacks and erroneous code. Thus, our proposal paves the way for large scale hosting of Grid or web services in commercial scenarios.

The paper is organized as follows. Section 2 presents the threats associated with native code in a Grid service environment. Section 3 discusses related work. Section 4 presents our approach to native code security. The changes required by our security approach to the development, deployment and execution of services are described in section 5. Section 6 presents the evaluation of our approach. Section 7 concludes the paper and outlines areas for future research.

2 Analysis of the Native Code Threat Model

For our threat analysis we consider a shared service hosting environment. The service-oriented ad hoc Grid is such an environment with the most demanding security requirements, since it allows the deployment of foreign services into a running service container. Threats may also arise in more statically configured Grid environments or even web service hosting scenarios. In both cases, a single service container can be used to host more than one service.

The focus of our investigation are the security threats arising from the native code contained within Grid or web services that are deployed into a Java based web service hosting environment. Security threats in the shared hosting environment arise from direct access to the underlying system or direct access to other service instances running in the hosting environment, that cannot be limited by Java security managers or sandboxing.

We distinguish between two types of attacks, the first focusing on data managed by the hosting environment or the other services, and the second one abusing other system resources, for instance network bandwith or CPU cycles. Examples for each of the resulting attack scenarios are presented in the following:

- Data attack against hosting environment: A malicious native service may be used to extract or alter security critical data from the underlying operating system or hosting environment such as the system password files, certificate files or service container configuration. In a common Grid environment, the native part of a service is executed with the user rights of the hosting environment, enabling the malicious service to read the configuration of the hosting

container, and in many cases even allowing the alteration of configuration files (such as the container authorization lists).

- Data attack against other services: A malicious native service may be used to read temporary data or results produced by other services as well as input data used by those other services. If, for example, a pharmaceutical company uses a Grid node for computations in the design phase of a new drug, a competitor may deploy a malicious service that extracts the experimental data used as input of the computation or the resulting outputs.
- Resource attack against hosting environment: A malicious native service may implement a spam bot that is used to send unsolicited bulk emails from the Grid service hosting network node. A number of denial of service attacks also fall into this category. By using native code, an attacker can cause the underlying operating system or hosting environment to crash, evading type safety mechanisms and sandbox constraints of a pure Java environment, effectively performing an internal denial of service attack on the service host.
- Resource attack against other services: A malicious native code may invoke methods from other services directly or use software licenses for 3rd party software that belongs to other service instances.
- Both of the resource attacks can be subdivided into illegal access to resources and denial of service against the local system through excessive resource consumption. Note that participation in a distributed denial of service attack counts as illegal resource access since the host is only used to harm other systems by illegally using the network interface, whereas recursively starting new threads is a denial of service attack against the hosting system.

3 Related Work

Different mechanisms for the protection of UNIX like operating systems such as Linux or FreeBSD, OpenBSD and NetBSD with respect to untrusted applications have been proposed. A very popular mechanism is the virtualization of the entire hardware, allowing a guest operating system to run in a virtual machine environment created by the host operating systems. Such virtual machine systems include Usermode Linux [4] or Xen [5]. The latter system has seen a great increase in popularity for the small performance overhead caused by its virtualisation technology.

Chroot confines file system access of a process run in the chroot environment to a different base in the file system. Some well known mechanisms exist for processes to break out of the chroot

environment and access files outside of this chroot jail. Those vulnerabilities have been addressed by the BSD implementation of the jail system call. Jails partition a BSD environment into isolation areas. A jail guarantees that every process placed in it will stay in the jail as well as all of its decendant processes. The ability to manipulate system resources and perform privileged operations is limited by the jail environment. The accessible file name space is confined in the style of chroot (i.e. access is restricted to a configurable new root for the file system in the jail). Each jail is bound to use a single IP address for outgoing and incoming connections, it is also possible to control what network services a process within a jail may offer. Certain network operations associated with privileged calls are disabled to circumvent IP spoofing or generation of disruptive traffic. The ability to interact with other processes is limited to other processes in the same jail.

Systrace [6] has become a popular mechanism for call restriction as well as privilege elevation on a fine grained scale without the need for running entire processes in a privileged context namely in OpenBSD and NetBSD with ports being available for Linux and FreeBSD as well. It uses system call interposition to enforce security policies for processes run under the control of systrace. Systrace is implemented in two parts, an addition to the kernel that intercepts system calls, comparing them to a kernel level policy map, disabling the call if a negative entry or no entry at all is present. The kernel level implementation is assisted by a user-level part that reads and interpretes policy specifications to hand them to the kernel level policy map, report policy enforcement decisions to the user applications and even call GUI applications for interactive generation of policies.

Janus [7] is one of the first system call interception tools. It uses the ptrace and /proc mechanisms which are claimed not to be a suitable interface for system call interception, since for example race conditions in the interface allow an adversary to completely escape the sandbox [8]. Janus has evolved to use a hybrid approach similar to systrace to get direct control of system call processing in the operating system [9].

The ability to set the effective user id of CGI programs to another user than the user id the calling web server runs under was introduced as the suEXEC capability in Apache 1.2 [10]. Our approach also offers the possibility of using setuid on the native processes, in addition other more fine grained access restriction methods may be used in a mixed Java and native code environment.

Emerging proposals for isolation of different Java threads in the same JVM address security

threats arising from the sharing of a single JVM between different applications [11]. An approach to implement such a shared JVM is actively pursued by Sun microsystems in the form of the multitasking virtual machine (MVM) [12]. While MVM offers mapping of isolated Java threads to operating system processes, only a prototypical implementation for Sparc Solaris has recently been published and would require the availability of the MVM for the hosting system and a switch to this new JVM. Porting a service hosting environment like the Globus Toolkit and Tomcat to the new VM requires substantial efforts to changes those systems [13].

The possiblity to decouple native processes from the process space of the Java virtual machine has been investigated in [14] in order to achieve better robustness of Java applications relying on native methods, not to enhance security of a shared Java environment or the underlying operating system. Systems like the Entropia Virtual Machine for Desktop Grids [15] or GridBox [16] propose the application of virtualisation and sandboxing technologies to achieve security for native Grid applications. They can only isolate entire native applications. Applied to the scenario of Java Grid services relying on native components, they cannot provide isolation of different services inside a single JVM.

4 An Approach to Native Code Security

Our security architecture addresses countermeasures for attacks stemming from native code used in Grid or web services, that fall into the different classes described in section 2. The technique used is process separation and confinement into secure sandboxes in order to allow for a flexible and fine grained definition of execution policies in an open multiple service environment. Our requirement does not require multiple instantiation of the JVM for isolation of different services, it is also independent of the JVM implementation allowing any JVM to be used to run the Grid or web service hosting environment.

The sandbox for Java classes within the JVM is defined by a security manager. Based on a given policy, the security manager controls access of Java classes to certain resources such as the file system or network interfaces. The Java security manager can block file system access for pure Java classes that must use the File classes of the Java IO packages for file system access. From an operating system perspective, all file accesses from the JVM is performed with the user rights of the owner of the JVM process. Child processes for native code also inherit the user ID of the JVM process. While a fine-grained and policy based restriction of resource access is possible for pure Java code by means of a custom security manager, this restriction of rights is impossible for native parts of a service. The JVM cannot keep the code from opening file handles with the permissions inherited from the JVM process.

Java offers two possible ways of using native code. The first is the creation of a new child process using Runtime.exec or ProcessBuilder.start, the second is the direct invocation of native method implementations through the Java Native Interface (JNI) [17]. The only means of protection against malicious native code in the standard Java language is the use of a security manager and a policy that prohibits execution of native code as a child process or loading of shared libraries (System.load, System.loadLibrary). This is not desirable in an environment like the service-oriented Grid, where reliance on native implementations can be expected to occur frequently.

4.1 Confinement of JNI Bound Implementations

The Java Native Interface specification defines the interface between the JVM and native methods implemented in C/C++. It enables invocation of native method implementations from Java classes and callbacks to Java methods from the native code. The JNI specifies a mapping from names of Java methods declared as native to C/C++ method names as well as mappings between Java types and native types. A sample method native int intTest(int i); in the class test.A would be mapped to the native method: Java_test_A_intTest. The first two arguments of this method are used to pass pointers to the JNI interface and the objects self-reference (this) to the native implementation. Followed by other parameters defined in the Java class for the native method.

The JNI interface is organized like a C++ virtual function table. It is passed by reference to the native implementation and managed by the JVM per thread (i.e. a native method may be invoked from different threads and therefore receive different JNI interface pointers, invocations from the same thread are guaranteed to pass along the same pointer). The structure itself contains a reference to an array of function pointers to implementations of the JNI interface methods. Besides passing an invocation result with return, the native method must use those JNI interface functions for access to any method or field in Java classes and objects managed by the JVM. The native methods are compiled into shared libraries and Java code using native implementations loads those shared libraries using System.loadLibrary. Native code is then executed in the process space of the JVM which leads to the serious threats described before. The native code cannot be further constrained on a fine grained per-service level, only confinenment based on the JVM process owner is possible. Figure 1 shows the relationship and confinement area using a standard approach for interfacing with the native code through the JNI.

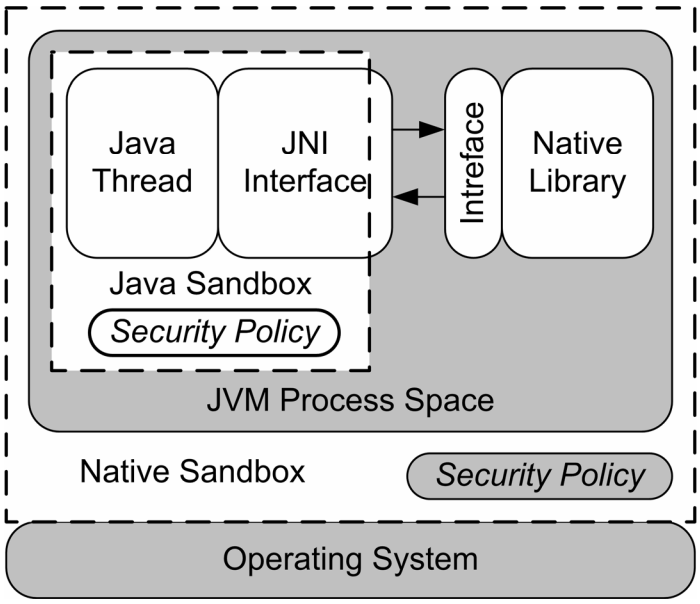

Figure 1: Standard access to native code through the JNI interface.

As a solution to this problem we propose the decoupling of the process spaces by use of an automatically generated transparent proxy intercepting all calls to the native implementation shown in figure 2. The native component of the service is replaced by a generated proxy that exposes exactly the interface of the original component. This proxy now receives all the calls to the native methods from the JVM. Such a call is passed on to the original native implementation that is instantiated in a different process than the JVM and managed by a process server. We refer to the wrapped and sandboxed native process as the I-Process. Creation of the I-Processes for the Java based Grid service hosting environment is managed by a custom process manager.

The process server acts like the JVM to the native method implementations, it passes a reference to an altered JNI interface implementation to the original native code. Every reference to the JVM from the original native code is thereby intercepted by the custom JNI interface implementation. The transparent proxy and the process server communicate by means of standard IPC or RPC mechanisms, depending on the security and functionality requirements.

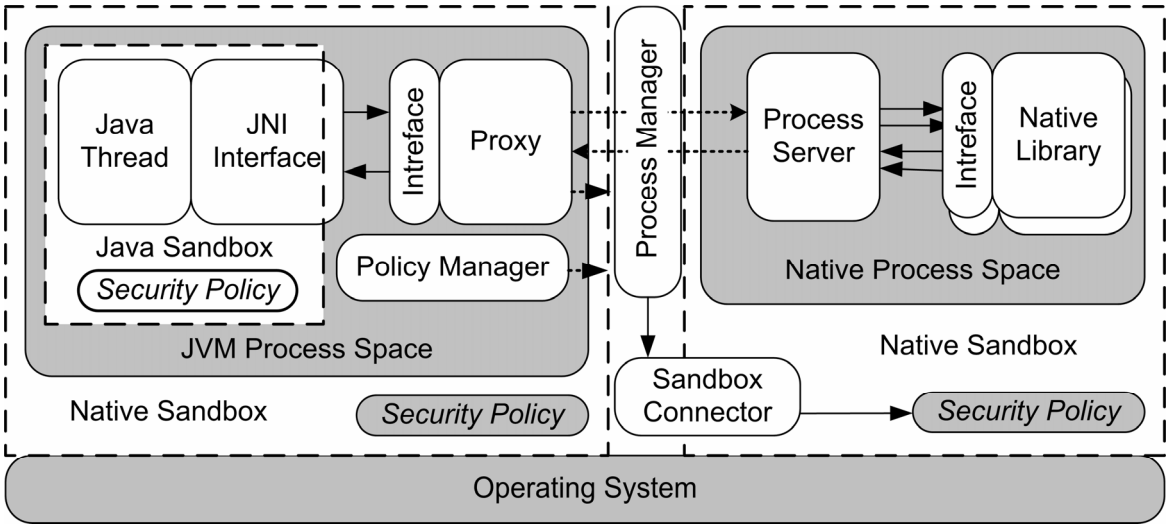

Figure 2: Decoupled process spaces for JNI attached native code, enabling secure isolation of native code.

4.2 Confinement of Execute Requests

In the previous section, native code was called via the JNI interface. A second possibility to execute native code from Java is to use the Java Runtime class to create a new shell environment where the native code is then run. The Runtime class uses the ProcessBuilder to create a new shell. The ProcessBuilder itself uses a statically linked JNI native create method to create the operating system shell. Unfortunately the proxy approach cannot be utilized in this case since the create method is integrated into the JVM. It is, however, possible to offer a custom ProcessBuilder which can sandbox a newly created shell on behalf of a service. This feature is currently not implemented in our proof-of-concept solution. To deal with this security issue in the meantime, we use the Java SecurityManager to forbid the ProcessBuilder to create new shells, thus restricting native code to be run via JNI.

4.3 Policy and Process Instance Management

In previous work [3], we have proposed a solution to intra-engine service security confining services or groups of services by use of a dedicated Java class loader. We extend this grouping scheme to the native parts of services, allowing the creation of sandboxes for I-Processes per service (or group). The hosting provider can attach a security policy to the service (group) that restricts resource access to the underlying operating system for all I-Processes created by services within this group. All processes started for services within the same service group also share the native sandbox. To be able to securely group different services, our solution to grouping Java services already provides an access control mechanism. A public-private-key pair is generated for a each newly created group. This key is obtained by the group owner (i.e. the creator of the group) who uses the private key to sign service archives containing the service

implementation (GAR files for the Globus environment). If the service implementation is signed using the correct private key, it is admitted to join the service group.

When a Java thread requests access to a native method, the transparent proxy implementation of this method will first of all check whether an I-Process has already been created for the Java thread. If a new I-Process has to be created for the current invocation, the group ID of the service is determined from the service class loader or its parent group class loader. This information can then be used to obtain the security policy that was specified for the service group by the hosting provider from the policy manager. This policy is passed along with the thread ID to the process manager that in turn creates either a new sandbox and an I-Process for the Java thread or only a new I-Process within an existing sandbox for the service group.

Creation of the sandbox environment is delegated to a sandbox connector. This component allows our isolation environment to be implemented for different underlying operating systems or different sandboxing techniques. It handles initial setup of the sandbox, inclusion of all necessary dependencies for a native library, installation and possible mediation of policies. The process manager also uses the sandbox connector for creation of new I-Processes as child processes of the sandbox connector, when such a new I-Process is requested by a Java thread calling into the transparent proxy. The hosting provider can specify the sandbox connector to use along with an access policy for a service group. Attachment of a security and sandbox policy to a service group may happen based on the user ID of the user that initially created the service group. Users may also be grouped allowing the application of a default policy to all service groups created by deployment of services by unknown users.

The Web Service Resource Framework introduces the notion of a so called Web Service Resource (WS-Resource) into the web service framework. A WS-Resource is the combination of a stateless web service and a state capturing Resource Property Document. A client receives a resource address upon creation of a WS-Resource for later reference in subsequent interactions with the WS-Resource. Service instances may attach to the resource property document in order to change the state data starting off from the current state of the WS-Resource. Isolation of the native service components connected to individual WS-Resource instances from each other would require the transparent proxy to identify the WS-Resource corresponding to the Java instance emitting the native call to the transparent proxy. There are many cases that prohibit the identification of the corresponding WS-Resource from the native side if no special precaution has been taken in the original library wrapper.

Such a fine granularity of confinement is only needed if the service implementation is untrusted and there is real concern that the service implementation could be used to exchange data between resource instances created for different users. A solution to this problem is the use of distinct service groups (and the corresponding class loaders) for different instances of the service. In this case, our proposal automatically isolates the I-Processes corresponding to different WS-Resources since they originate from different class loaders.

Our proposed solution can so far protect the hosting environment and other services against data attacks as well as illegal resource access. Depending on the sandbox capabilities, it is also possible for our process manager to monitor resource utilization and constrain e.g. CPU time as well as disk space or network bandwidth used by individual sandboxes. This still allows services to perform denial of service attacks against other service instances running in the same sandbox but leaves the option of shutting down entire sandboxes when they show behaviour that can cause harm to the hosting environment. A problem still remains in effective resource management in the JVM. No widespread solution to the problem of monitoring and managing resource consumption of individual Java threads (belonging to a service instance) is currently available.

4.4 Secure Process Spaces

A number of different options for the secure execution of native code (i.e. enforcement of access restrictions to the host operating system and isolation of processes against each other) can be used in our architecture. Those approaches have been introduced in section 3. A balanced decision needs to be made between the cost (i.e. instance creation time, computational overhead, increased resource consumption) incurred on the original Java service hosting environment and the strength of security offered by the chosen method. We will now discuss some of the implications of using the different techniques for native code isolation, with the cost and complexity imposed by such systems and the security provided by them.

Dedicated or Virtual Hosts can be used to achieve a very high level of confinement since they add a layer of abstraction for the entire hardware. Instantiation of the native process requires the shared object file to be present in the file system accessible by the new operating system instance that runs the I-Process. While startup times for the guest operating system instances of a virtual host environment can be somewhat leveled by pre-loading a number of guest operating systems that are used on demand, the memory consumption of this method is very high. We therefore favor a more fine grained and lightweight approach to resource virtualization or

compartmentalization of the I-Processes.

Changing the effective user ID of the I-Process provides security based on standard file and resource access control of the operating system. While this method requires no preparation of the sandbox and imposes virtually no performance overhead, it offers only very limited security against exploitable weaknesses in the operating system. A downside of this approach is the need for a pool of user accounts that I-Processes are mapped to by the sandbox manager.

BSD Jails are a way to partition a BSD environment into isolation areas. The jail system call has been developed to explicitly counter well known techniques to escape a similar chroot environment. The overhead created by running a process in a jail is very low. Jailed processes are tagged to belong to a certain jail, the system enforces security by identifying this tag for certain privileged operations, resulting in a very small overhead for only those calls limited by the jail environment.

Systrace offers even more fine grained policy enforcement over processes running under control of systrace. It has become a very popular tool for privilege control for a number of BSD variants. We experienced acceptable performance overhead with a Linux port of systrace. For a first prototypical implementation of the system, we employ a combination of chroot and an extended implementation of systrace as they offer the best balance between overhead and gained level of security. Shared objects and libraries they depend on are either copied mapped by hard linking into the system and can be protected by using systrace to intercept write attempts on the libraries. The (small) memory overhead is limited to the process and sandbox management components.

5 Development, Deployment and Execution

In this section, we discuss changes that are required by our security approach to the development and deployment process as well as special concerns during the execution of services.

5.1 Development

Using our system does not strictly require changes to the development process of the service containing native components. There is, however, added benefit for service developers and service users in taking a post-development step: specifiying a requirement profile for their service. Such a requirement profile can also be derived automatically at deploy time, but specifying the profile beforehand helps the platform to compare requirements against the security policy and determine mismatches that lead to failure of service instantiation or

execution.

The requirement policy can either be specified manually or automatically generated in the following way: Our addition to the hosting environment supports a trace mode that can be used by the developer to install, instantiate and use the newly developed service. The process manager advises the sandbox manager to create the sandbox environment in trace mode that records any resource access of the service instance running inside this sandbox. Since we can assume the developer to trust his or her service implementation, no restrictions are enforced on the service instance. The recorded requirements stemming from a trace run can then be transformed into a generalized requirement profile to be used in the deployment descriptor of the service package. This consolidated policy contains generalizations of libraries that the service depends on, files that the service accessed and network addresses the service connected to. The consolidation operation is used to sort out typical calls that stem from the generated wrappers and standard library handling methods, in order to make the process of customization easy for the developer. Again, the requirement policy does not affect security but performance when searching for a compatible node that can be used to deploy and run the service.

5.2 Deployment

Deployment of a service containing native code consists of the following steps:

- Compare the service requirement policy with the security policy specified by the hosting provider (optional)
- Generate a custom transparent proxy for the native components of the service
- Create or join a service group and bind the security policy to the group
- Pre-load secure sandboxes (optional)

These steps must be executed on the target machine since the generated code runs in the privileged hosting environment and as such must not be supplied by the service.

The first step during deployment checks whether the environment into which the service is to be deployed offers sufficient access rights to successfully run the service. Since creating requirement specifications and publishing security policies creates additional costs and is not always required, this step is optional although it is recommended. This allows manual decisions to be made about which machines are suitable. If a service does not specify what requirements it

has, it can be deployed onto any machine, but will fail to execute if any operation is attempted that is not permitted by the secure hosting environment.

Next, for all shared objects a transparent proxy is generated by introspectively analyzing the shared objects to discover which methods contained therein are JNI compliant. Based on this information, the Java classes can be reflectively analyzed to retrieve the method signature since it is not contained in the shared object. From that, a proxy capable of accepting all pre-defined native calls in place of the original shared object is generated. The proxies are registered with the hosting environment to enable run-time monitoring of the sandbox integrity (see section 5.3). This step also includes resolution of dependencies (i.e. identification of other shared objects the service requires to run). These dependencies are recorded and attached as a sandbox descriptor to the transparent proxy.

If it is necessary that different services containing native code interact directly (i.e. not via their Grid service interfaces), the services must be deployed into the same group so they are not separated by a sandbox. The first service creates a group and the hosting environment binds a policy file to the group based on the user id of the service deployer. The service deployer also gets a public/private key pair with which all other services which should be permitted to enter the group are signed. Only properly signed services may enter the group.

The last step of the deployment process prepares the sandbox for operation. This can include the booting of a virtual hosting environment or making certain files accessible inside the sandbox (e.g. include copies or hardlinks of required shared objects in a chroot environment). This step is optional since it uses up system resources and should only be executed if the service requires quick first response times.

The steps required in the development and deployment process are visually summarized in figure 3.

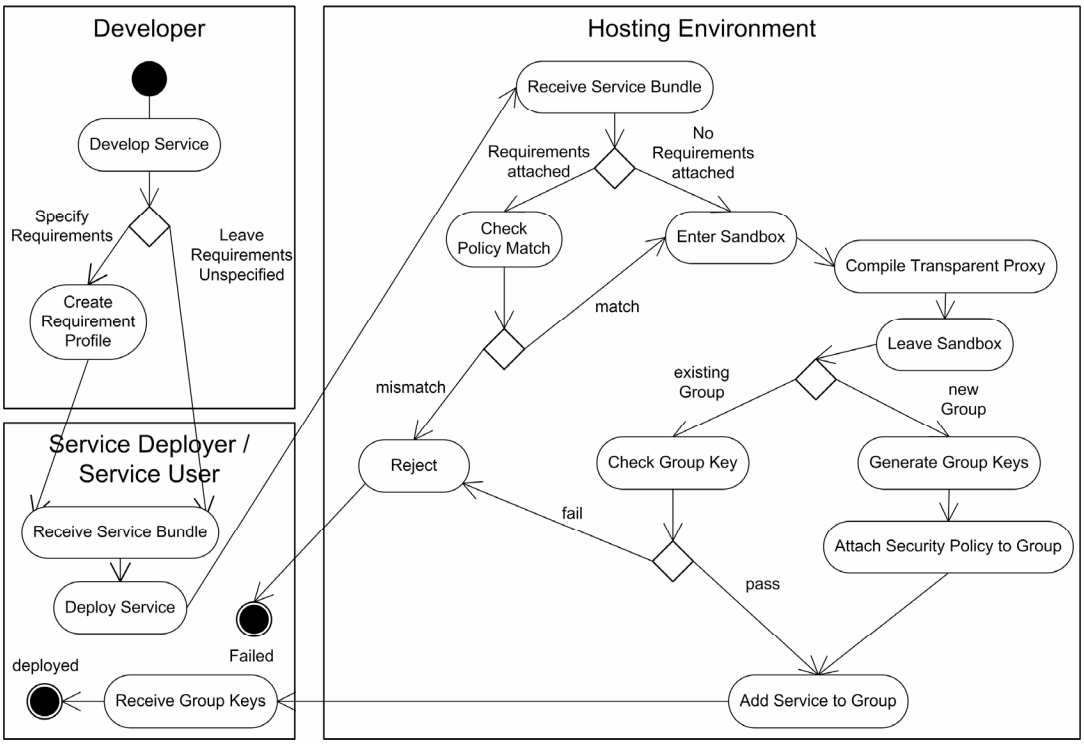

Figure 3: Steps in the development and deployment process. (The service bundle is signed with the group private key for deployment into an existing group.)

5.3 Execution

For most services containing native code, the environment is now fully configured and the service is sandboxed. As mentioned above, the proxies responsible for the sandboxing are created through code introspection. This creates the following security risk: If a service contains a shared object in an obfuscated form or generates code on the fly, the introspection process used during deployment will not be able to generate the sandboxing proxies and thus the sandbox would not be safe. To combat this, the Java SecurityManager is extended to check at runtime whether libraries which are to be loaded were processed during deployment and thus the needed proxies were generated. If there is no registered proxy for a given library, it is generated at run-time and substituted for the original shared object. Since only very few legitimate applications require the dynamic generation of code, a warning is sent to the hosting environment informing the administrator that code is being run in the sandbox which did not enter through the deployment process. This code is sandboxed but it should nevertheless be checked by the administrator.

6 Evaluation

Two main factors govern the performance of the presented security solution for individual calls to native methods: The overhead imposed by the native sandbox of the operating system and the

overhead imposed by the process dislocation technique that requires local inter process communication.
The cost analysis for native code sandboxing is subject to evaluation by the creators of the various techniques offering a solution in this area and is not covered by the scope of this paper.

In order to obtain an initial result on the performance of the presented solution, the following experiment was conducted to assess the overhead of transparent process dislocation for native code used in a Java service implementation through the JNI. A Java test class was created that uses a native method implemented using C. The absolute time before and after an invocation of the native method in the Java class was measured in a loop of 500 invocations. In the first case, the original native library compiled using gcc 4.0.1 was used. Then, the native library was replaced by a transparent proxy library that used RPC communication to perform method invocation on the original library in a process separate from the JVM. Again, the runtime of the native method invocation in the Java class was measured.

The regular call took 3 microseconds on the average while the call using separate process spaces took 566 microseconds. This increase in time required to perform a native call by a factor of 182 is only acceptable if relatively few native calls are required from the Java code into native libraries. Forutantly this is the normal situtation when a Grid service implementation provides a front end to functionality provided by native business or scientific libraries. In this case methods provided by the library are used on a macroscopic level and the service will typically spend substantial amounts of its runtime in the native code alone instead of requiring many native method invocations in the Java implementation.

A benefit of the micro jailing technology lies in the small memory overhead created by the solution. The native process manager and all RPC related components require less than 1 megabyte of main memory. This is a small overhead compared to process isolation by use of dedicated Grid service container instances for different services and users. Using these dedicated Grid service containers, a complete JVM must be instantiated, requiring at least 20 megabytes of memory.

7 Conclusions

In this paper, we have analyzed the threat scenarios that emanate from native code in a service-

oriented ad hoc Grid environment and categorized them into four distinct types: data attack against the hosting environment, data attack against other services, resource attack against the hosting environment and resource attack against other services. To counter those threats, a novel security architecture was presented, which enables confinement of native components of Grid applications into a secure environment. The security framework is based on dynamically created JNI proxies which create a pipe between the secure Java environment and the secure native environment. The security solution is capable of protecting the hosting system as well as services from each other. While our work is focused on the Grid environment, our security solution also offers benefits in a regular shared web service hosting environment.

Future work includes the extension of the policy generation and matching mechanism and further usability tools. A custom ProcessBuilder will be provided to extend the sandboxing capabilities of the system to include Runtime.exec() commands as well as JNI calls.

Acknowledgements

This work is supported by the German Ministry of Education and Research (BMBF) (D-Grid Initiative, In-Grid Project)

References

[1] Smith, M., Friese, T., Freisleben, B.: Towards a Service-Oriented Ad Hoc Grid. In: Proc. of the 3rd International Symposium on Parallel and Distributed Computing, Cork, Ireland (2004) 201–208

[2] Friese, T., Smith, M., Freisleben, B.: Hot Service Deployment in an Ad Hoc Grid Environment. In: Proc. of the 2nd Int. Conference on Service Oriented Computing, New York, USA, ACM Press (2004) 75–83

[3] Smith, M., Friese, T., Freisleben, B.: Intra-Engine Service Security for Grids Based on WSRF. In: Proc. of Cluster Computing and Grid, Cardiff, UK (2005)

[4] Dike, J.: User-Mode Linux. In: Proc. of the 5th Annual Linux Showcase and Conference, Oakland, USA (2001)

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proc. of the ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, USA, ACM Press (2003) 164–177

[6] Provos, N.: Improving Host Security with System Call Policies. In: Proc. of the 12th USENIX Security Symposium, Washington, USA (2003) 257–272

[7] Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A Secure Environment for Untrusted Helper Applications. In: Proc. of the 6th Usenix Security Symposium. (1996)

[8] Wagner, D.A.: Janus: an Approach for Confinement of Untrusted Applications. Technical report, CSD-99-1056 (1999)

[9] Garfinkle, T.: Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In: Proc. of the ISOC Symposium on Network and Distributed System Security. (2003)

[10] The Apache Software Foundation: (Apache HTTP Server Documentation: suEXEC) `http://httpd.apache.org/docs/suexec.html`.

[11] Sun Microsystems, Inc.: Java Specification Request 121: Application Isolation API Specification (2003)

[12] Czajkowski, G., Daynès, L., Titzer, B.: A Multi-User Virtual Machine. In: Proc. of the USENIX 2003 Annual Technical Conference, San Antonio, USA (2003) 85–98

[13] Jordan, M., Daynès, L., Czajkowski, G., Jarzab, M., Bryce, C.: Scaling J2EE(TM) Application Servers with the Multi-Tasking Virtual Machine. Technical report, SMLI TR-2004-135, Sun Microsystems Laboratories (2004)

[14] Czajkowski, G., Daynès, L., Wolczko, M.: Automated and Portable Native Code Isolation. Technical report, 2001-96, Sun Labs (2001)

[15] Calder, B., Chien, A.A., Wang, J., Yang, D.: The Entropia Virtual Machine for Desktop Grids. In: Proc. of the First ACM/USENIX Conference on Virtual Execution Environments. (2005)

[16] Dodonov, E., Sousa, J.Q., Guardia, H.C.: GridBox: Securing Hosts from Malicious and Greedy Applications. In: 2nd International Workshop on Middleware for Grid Computing. (2004)

[17] Sun Microsystems, Inc.: Java Native Interface Specification (2003)