Tolerating Intrusions in Grid Systems *

Luis Sardinha Nuno Ferreira Neves Paulo Veríssimo

Departamento de informática
Faculdade de Ciências
Universidade de Lisboa
Portugal

Abstract

Grid systems are designed to support very large data set computations, that potentially access significant resources spread through several organizations. These resources can be very tempting for a hacker because they can be used, for example, to break pass-phrases with brute-force attacks or to launch distributed denial of service attacks to a given target. In this paper, we explain how malicious intrusions can be tolerated in the Globus toolkit grid environment, a de facto *standard in this area.*

Keywords: Grid Security, Intrusion Tolerance, Replication, Globus.

1 Introduction

Grid systems coordinate resources belonging to several organizations in order to support the execution of large applications. These resources might include a number of (parallel) computing systems and massive storage devices. They are usually connected by networks with reasonable bandwidths because many times they are located in more than one site. The Globus Toolkit is framework developed for this type of environments, that provides a number of services that can be used for the construction of grid applications or other programming tools [4]. Due to its success, it is considered a *de facto* standard for grid computing.

Current experience shows that it is very difficult (or impossible) to build a complex system completely secure. Since grid resources are quite attractive, one should expect to see these systems become a target for the hacker community. For example, the considerable computing power can be used to break passwords, the storage devices can save hacker tools and other information, and the large bandwidth networks are ideal for denial of service attacks. Therefore, a design principle that one should follow when developing grid systems is to assume that eventually some of them will be intruded. Then, one should build them is such a way that attackers are forced to spend a significant effort in order to subvert the system.

The Globus Toolkit places a computer in the border of an organization's network that serves as the interface between the exterior and the interior systems (we will call this machine *Globus host*). This host makes, among other things, the mapping of an outside user to an inside one. The mapping is necessary because each organization has complete control over its users and the mechanisms employed to enforce the security policies (e.g., one organization might have password files, while other could use Kerberos). Therefore, in general, if an external user wants to access an internal resource, she (or he) will have first to obtain the local credentials that will authorize the desired operation.

*This work was partially supported by the FCT through project POSI/CHS/39815/2001 (COPE), and the Large-Scale Informatic Systems Laboratory (LASIGE).

To avoid any security flaws, the Globus Toolkit was carefully designed and implemented. However, even if we assume that Globus is completely secure, which is very difficult to prove, an attacker may explore any other vulnerability to enter in the Globus host and, for example, perform an impersonation attack by changing the mapping configuration files. Whenever this happens, she will be able (or anyone else she wants) to become any of the inside users managed by Globus. Consequently, just by breaking into one computer, an attacker will have access to all available grid resources.

In this paper we explain how to tolerate intrusions in the Globus Toolkit version 3. Our solution integrates a replication scheme with the authentication and authorization process, in such a way that even if the Globus host is compromised, the organization's batch systems can not be immediately used by the attacker. However, to achieve it, all inner nodes must be capable of verifying a permission ticket (a credential issued by the Intrusion Tolerant Authentication and Authorization System (ITAAS)). The administrator can also manage the ITAAS policies in a coherent and secure way, and configure all nodes giving the replicas' identification and public keys.

2 Background: Globus Toolkit

The Globus Toolkit can be used to interconnect several physical organizations in order to create a scalable, dynamic and distributed virtual organization composed by individuals that seek to share and use diverse resources in a coordinated fashion. The third version of Globus Toolkit (GT3) is an implementation of the Open Grid Services Infrastructure (OGSI) [8], a technical specification of the concepts described in Open Grid Services Architecture (OGSA) [4]. The OGSA defines a new common and standard architecture for grid-based applications, generally called Grid Services, which are an extension of a Web Service.

Globus Toolkit's previous versions use, for the communication, a set of proprietary protocols with specific syntax and ports (potentially creating some difficulties with the firewalls of the institutions), which introduced several portability problems. In this new version, the communication protocols are not prescribed, but in the current implementation they are based on SOAP [1]. It provides a means of messaging, using XML envelopes to encapsulate payloads, where HTTP is the most commonly used underlying protocol. The Web Services Description Language (WSDL) [3] is utilized to describe the methods of a Web Service, and it provides a way for expressing operation signatures and bindings to protocols. The message-level security support is based on the WS-Security [5], with XML-Signature and XML-Encryption standards.

The GT3 Core security infrastructure is based on the Java Authentication and Authorization Service (JAAS) framework. It allows Java Grid Services to remain independent from underlying authentication mechanisms.

3 Intrusion Tolerant Authentication and Authorization System

The authentication and authorization process of GT3 is performed in two steps: the first one is made by the GT3 core, where it verifies the credentials (e.g., a proxy certificate) given by the client. Each GT3 has a group of trusted certificates, each one assigned to a different user (or machine). Whenever a request and associated credential needs to be validated, the GT3 core first makes sure that it was issued by an entity with a trusted certificate. Then, it verifies if there is a mapping of the external client to a local user using a configuration file. In order to scale this file, the administrator can group several users and map them into the same local user (e.g. all individuals from organization A are mapped to user org_A).

The second step is executed by the target service, and it confirms if all security requirements are met. GT3 provides a declarative approach to customize the security policies of a service. Basically, services have the flexibility of defining their own policies, where they can specify for instance that certain messages need to be signed while others have to be encrypted. Whenever a request arrives, the service has to ensure that it satisfies all conditions imposed by the security policy.

If an adversary gains access to the Globus host,

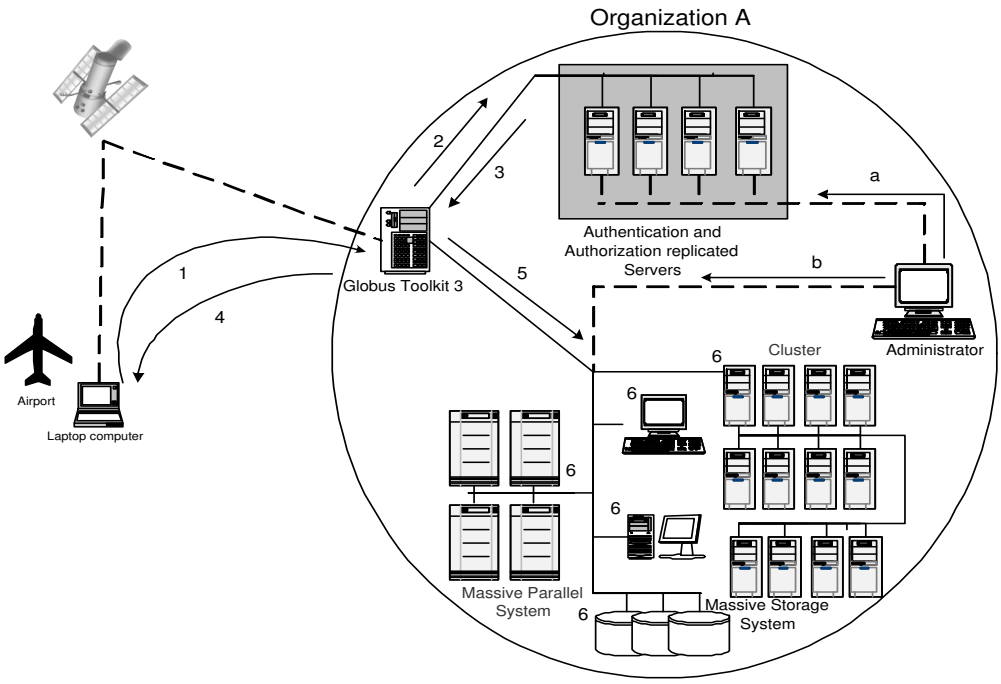

Figure 1. An example of a ITAAS architecture.

then she can perform several different types of malicious attacks. For example, she can change the security policies of a service or map anyone to almost any local user (normally used by Globus), and the inner systems will not notice the difference. After performing these actions, the attacker can use the local infrastructure to execute jobs on the machines managed by Globus with the same permissions as any of the authorized users.

To prevent this type of attacks, we have replicated the authentication and authorization system with a protocol capable of tolerating intrusions. An adversary will have to penetrate f out of a total of $n \geq 3f + 1$ machines to compromise the system, where f is a configurable parameter managed by the administrator. We designed this solution respecting the OGSI specification (which means that it can be adapted to any other of the organization's security infrastructures) because the core of our model is implemented inside GT3 and only a permission ticket (described below) is seen outside of it.

3.1 The protocol

In this section we describe how the authentication system will work. For a better understanding, the numbers in Figure 1 illustrate the following steps of the protocol.

1. An external client (we will call client to any user or service working on behalf of a user) contacts a service stored in a service container of the Globus host. The client can be located in any place, as long as she has Internet access.

2. Globus preforms a few initial validations, and then it sends an authentication and authorization request to the replicated servers – the ITAAS replicas. This request contains the credentials forwarded by the client to the Globus host.

3. Each replica consults its configuration files to determine if the client has the right to use the Globus resources. In the affirmative case, it creates a reply containing the following information: a unique sequence number, a life-

time for this authorization, the client certificate, and the associated internal user identification. Otherwise, an error code is returned explaining the reason why the client is not authorized to use the system. The reply is signed by generating an hash of the message and by encrypting it with the private key of the replica.

4. The authentication service located in the Globus host waits for the arrival of $f+1$ equal responses from the replicas (if we assume that a hacker can only compromise f replicas, then this ensures that we will be using a correct response). Then, it uses the information contained in the message to create a permission ticket. The ticket has several fields, and among them are: the sequence number, a lifetime, the client certificate, the client internal identification, and the $f+1$ signed hashes (properly identified).

   The permission ticket's lifetime usually is the same as the one in the credentials of the request, however, it can be shorter.

   To avoid overloading the ITAAS servers, the permission tickets are stored in a cache so that they can be re-used by other requests from the same client.

5. Globus may give some feedback to the user if everything went well (e.g. to advance the service's protocol) or return an error message.

6. When the service wants to use the local resources, it has to provide the expected arguments and configuration information, and the permission ticket.

   A resource uses the permission ticket to validate the requests made by the service. First, it verifies if the ticket is good (e.g., correctly signed by $f+1$ replicas), and then it uses the client internal identification to perform all access control decisions. If for some reason one of the validation tests is not satisfied, the resource returns an error and the request is aborted.

   To improve performance, resources can also create a cache of valid and already seen permission tickets. Whenever a permission ticket is received, the resource avoids most of the validation tests, simply by looking for the ticket in the cache.

The protocol makes the life of an adversary significantly harder before the whole system becomes compromised. After a successful intrusion on the Globus host, the hacker has basically no access to the resources of the organization because she does not have the necessary permission tickets and is unable to create them (the cooperation of the ITAAS servers is required for the ticket generation). At most, she can collect the permission tickets stored in the globus host cache and use them. However, since tickets are associated to specific users and are only valid during a period of time (specified by lifetime field), they are of limited interest to the adversary. If the adversary wants to use the resources of the organization, she will have to continue to attack more machines, either the ITAAS servers or the internal nodes. Only after compromising $f+1$ ITAAS servers, she will be able to create her own tickets.

The administrator of the system has basically to perform two tasks to manage the ITAAS server (see Figure 1):

a *Update information about external users:* Occasionally, the information about the external users has to be updated in the ITAAS servers (e.g., a new user is created). The administrator sends these updates to all replicas in a signed message. It is assumed that the private signature key is only known to the administrator, and that it can be protected from disclosure (e.g., by saving it in a chip card).

b *Update information about ITAAS servers:* A node can only validate the information produced by the ITAAS servers if it knows: the total number of replicas and their public signature keys. This information has to be updated in a secure way whenever the ITAAS configuration changes. To achieve this objective, updates are signed with the private key of

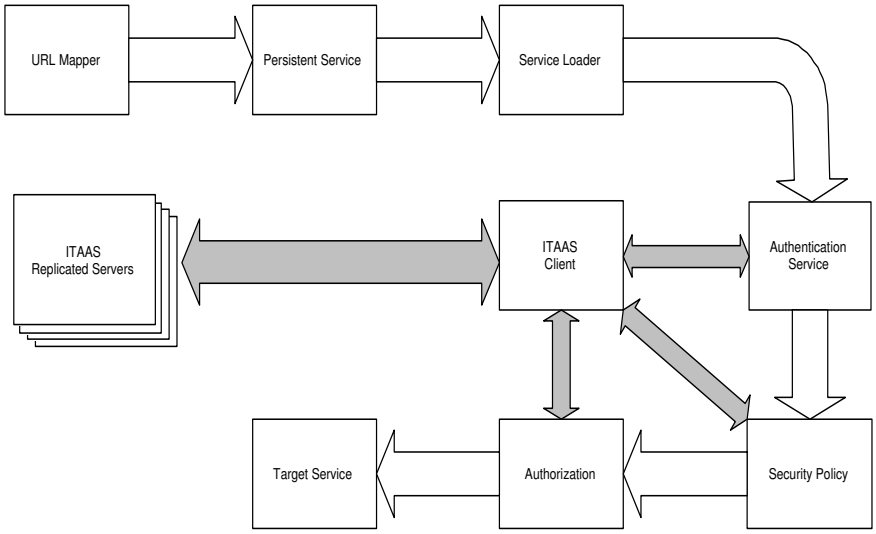

Figure 2. A simplified request dataflow

the administrator.

The administrator can exchange security for cost by keeping a different number of ITAAS replicas. This trade off, however, does not have to be made in a static way, since it is relatively simple to add more servers. Basically, one only needs to add a machine, copy the current state from a good replica, and update the information about the ITAAS servers (as described above).

3.2 Detailed View of a Request Processing in the Globus Host

When a request arrives at the Globus host it is processed through several handlers. These handlers are used to get the target service from the message header, load the target service, verify the authentication elements and the user's permissions, etc. These handlers, and their invocation order, are defined in a configuration file, which makes the replacement of any handler a simple operation, without requiring the change of any source code.

In our approach, we replace some of the authentication and authorization handlers by others that have the same functionality, but that do not rely on the local information. The handlers get the relevant authentication and authorization information through the ITAAS client – a small agent that communicates with the ITAAS servers.

Whenever a handler is called to process an external user request, it contacts the ITAAS client to obtain the necessary information. Then, the ITAAS client calls the ITAAS server to collect the $f+1$ responses, and to create the permission ticket. Using the ticket, it can then provide a reply to the handler.

The ITAAS client stores in a cache the permission tickets. These tickets can be reused if the same external client sends other requests to the Globus host, improving the performance of the service and at the same time reducing the load on the ITAAS servers. Since the cache must have a limited size, whenever it is full, the new entries will have to replace the older ones. Nevertheless, by adjusting the cache size, the administrator can reduce to a minimum the number of cache misses.

To avoid some denial of service attacks, the ITAAS client always validates the credentials given by the external clients, before sending any request to the ITAAS servers (e.g., if the current time is within the validity period). Moreover, it keeps in a cache all credentials that were refused in the recent past.

Figure 2 displays how our solution is integrated in the Globus context. Some of the handlers were removed from the figure because they are not rel-

evant to the scope of this paper. The white arrows show the various steps executed during a request processing. In the security related handlers some calls are made to ITAAS client that possibly has to communicate with the ITAAS servers (grey arrows).

4 Implementation

A prototype with our intrusion tolerant architecture has been developed for the current version of the Globus Toolkit. The implementation of the replicated ITAAS servers utilizes a generic replication library called BASE [6].

4.1 BASE

BASE provides a set of services that facilitate the development of replicated servers. With BASE, it is possible to replicate services that perform arbitrary computations provided that they work in a deterministic fashion, i.e., they must produce the same output when they process the same sequence of operations. Whenever this condition is not verified, which happens with most off-the-shelf implementations, it is necessary to use some conformance wrappers and abstract state conversions.

The core of the BASE implementation is a protocol for secure management of replicated services [2]. This protocol guarantees that all replicas receive the same requests from the clients, and that they process these requests in the same order. Since correct replicas execute the computations in a deterministic way, they all produce the same answers. Therefore, a client can use a simple voting scheme to remove the bad answers generated by malicious replicas (i.e., that are controlled by an adversary). The protocol returns good answers if the number of malicious replicas f is smaller than one third of the total number of servers ($n \geq 3f + 1$), which is optimal for this kind of systems.

5 Example of a Batch System - Condor

We will use a well-known batch queueing system, Condor [7], to exemplify how our solution can be integrated with this type of platforms. Condor is a specialized workload management system for compute-intensive jobs. Users submit their serial or parallel jobs to Condor, and then it runs them in the available computing nodes. It can be utilized both to manage a dedicated cluster of nodes or to harness wasted CPU cycles from otherwise idle workstations. Currently, Condor does not support job submissions through GT3. However, this situation is expected to change because Condor-G is fully interoperable with resources managed with previous versions of Globus.

Condor is comprised of a specialized node called the *central manager* and an arbitrary number of other machines that are members of the computing pool. The central manager acts like a repository of information about the current state of the pool. The Condor architecture has several kinds of daemons spread through the nodes submitting the jobs, the manager and the machines in the pool. For a better understanding of the system, we will briefly describe the most relevant daemons:

- Submit node: *condor_schedd* represents users to the Condor pool, and contacts the manager to find available nodes to submit jobs. There is one *condor_shadow* for each currently running job. It makes all decisions related to the job, which includes log the progress and service all "remote system calls" (they are redirected from the remote node).

- Pool node: *condor_startd* is responsible for enforcing the policy under which remote jobs are started, suspended, resumed, vacated, or killed. When *condor_startd* is ready it spawns the *condor_starter* to execute the job. *condor_starter* sets up the execution environment, starts the job and monitors its progress. When it detects that the job is completed, it sends back status information to the corresponding *condor_shadow* and exits.

- Central manager: *condor_collector* gets all information related to the status of the pool that is periodically sent by all other daemons. *condor_negotiator* gets the information about the available nodes from the *condor_collector*,

and tries to match the submitted jobs to the machines that will serve them.

One way to integrate the Condor in the architecture presented in Figure 1 is to use it to manage a cluster of nodes where external users can start jobs. In this case, there would be a single submit node located in the Globus host, a central manager in one of the machines of the cluster, and a number of pool nodes where the jobs would be executed.

When an external user wants to run a job, she must contact the Globus node, and execute the corresponding service. This service performs the authentication procedure, and obtains the permission ticket associated with the user. Then, it calls the local *condor_schedd* and, besides giving all information related to the job, it hands over the permission ticket. *condor_schedd* contacts the *condor_collector* and *condor_negotiator* daemons at the manager node to find out where the job can be started. The manager daemons, however, will only provide this information if the request carries a good permission ticket. Next, *condor_shadow* is spawned by the *condor_schedd* in the Globus host, and receives the job configuration data plus the permission ticket. *condor_shadow* contacts the *condor_startd* daemon of the assigned pool node (or nodes), and requests the execution of the job, giving the permission ticket. If the ticket is valid, a *condor_starter* is initiated to manage and run the job. In the end, the result of the job execution is returned to the Globus host, and provided to the user.

6 Conclusion

This paper explains how intrusions can be tolerated in Grid systems managed with the Globus Toolkit. Our solution is based on the secure replication of the authentication and authorization service of Globus in a set of servers, and on the introduction of a new permission ticket. A prototype of this solution has been developed for Globus Toolkit version three, and implemented with the BASE replication library. The paper also describes how the Condor batch system could be integrated in our prototype.

References

[1] Simple object access protocol (soap) 1.1, May 2000.

[2] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, February 1999.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, March 2001.

[4] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Global Grid Forum*, June 2002.

[5] Microsoft IBM and VeriSign. Web services security language(ws-security), April 2002.

[6] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, volume 35, pages 15–28, Banff, Canada, October 2001.

[7] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[8] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open grid services infrastructure (ogsi), June 2003.