Security Architecture in GAIA/2k

1 Introduction

Security concerns in GAIA can be classified among the following categories

- Authentication.
- Access Control.
- Secure Dynamic Loading of Components.
- Secure Tracking.
- Secure Boot-Strapping.

In the following sections we describe each in detail.

2 Authentication

Authentication consists of validating a users claim regarding his identity. This functionality is provided by the Authentication Service (AS). The authentication service provides different kinds of credentials to solve the various problems associated with it. These credentials enable the Access Control Service to provide Discretionary, Mandatory and Role Based Access Control.

2.1 The problems

The different kinds of problems which have to be addressed are

- **Delegation of authority to trusted program** : A user has to provide a program/service a mechanism by which the program/service can execute on the users behalf.
- **Delegation of authority to untrusted program** : If the user does not trust a program he might not want to give it complete authority to execute on its behalf. For example, Service S might not be trusted and instead of giving it complete authority a user might want to give it only the authority to user resources in `/space/DCL/3234` on his behalf.
  The Confused Deputy problem also manifests itself. This problem arises when a service/process runs with authority stemming from two sources. It is sometimes necessary in such cases to limit authority stemming from a particular source only to some resources.

- **Simple authentication** In some cases a user might just want to authenticate himself to a service. The service doesn't need to execute using the users authority. In such cases the user should be able to issue a credential which the service cannot misuse to execute on the user's behalf.

2.2 Credentials

The different kinds of credentials which are provided are:

- Generic Credential.
- Restricted Credential.
- Non Delegatable Credential.

2.2.1 Generic Credential

These credentials solve the first problem mentioned in Sec 2.1. A typical credential looks as shown below.

```
{
USERID = (bsgill)
ROLES = (student) (ACM member)
ATTRIBUTES = (Age = 23) (Field of Study = Computer Science)
}signed by AC
```

Figure 1: A Credential

These credentials give the holder of the credential all privileges. As a user might be a part of many roles and might have many attributes a list of roles and attributes for which the credential is valid is also sent.

2.2.2 Restricted Credentials

When the *DELEGATION RESTRICTED TO* field is present, the credential is termed as a restricted credential.

```
{
USERID = (bsgill)
ROLES = (student) (ACM member)
ATTRIBUTES = (Age = 23) (Field of Study = Computer Science)
DELEGATION RESTRICTED TO = (/resources/tmp) (/resources/space)
}signed by AC
```

Figure 2: A Restricted Credential

The restricted credential solves the second problem mentioned in Sec 2.1.

2.2.3 Non-Delegatable Credentials

These are credentials that can be used by the client to prove its identity to the server without giving any of its own rights to the server. This is achieved by a Non-Delegatable Credential issued by the *Authentication Service*. This credential has the IOR of the *TARGET SERVICE* that it authenticates the client to. Thus, the service cannot use this credential to contact other services pretending to be the client, as the other service will find out that the IOR contained in the credential does not match its own.

The credential also has a time field that stores the time when the credential was issued. Typically, these credentials will be used right after they are generated. So, to make the mechanism more safe, especially from replay attacks where the server is replaced by another server but with the same IOR, we use a timeout for these credentials. A sample Non-Delegatable Credential looks as follows.

```
{
USERID = (bsgill)
ROLES = (student) (ACM member)
ATTRIBUTES = (Age = 23) (Field of Study = Computer Science)
TARGET SERVICE = <IOR>
TIME = <time>
} signed by AC
```

Figure 3: A Non-Delegatable Credential

2.3 Credentials and Roles

Credentials and Roles are closely tied together. If a user logs in and chooses not to activated any role he gets only the credential as a user ($\{\text{user}\}_{signedAS}$). However, if he logs in and decides to activate Roles A and B, then his credential also indicates that he has priviliges belonging to Roles A and B ($\{\text{user,Role A, RoleB}\}_{signedAS}$).

When a program gets a credential it executes with the full rights of the user, roles which the credential identifies unless it is a Restricted Credential. Also at any time it should be possible for a user to change his credential so as to deactivate existing roles or activate additional roles.

When a user starts a program and wants to give the program his credential he should be able to specify that the credential be given only for specific roles.

2.4 Interface for Authentication Service

The CORBA interface for the Authentication Service is shown in Fig 5.

getCredentials returns a generic credential on being passed a matching pair of user and password.

To get a restricted credential **getRestrictedCredentials** is invoked with an existing credential and a string which represents the restrictions.

```
interface AuthenticationService
{

typedef sequence<octet> SignedCredential;
typedef sequence<string> StringList;

SignedCredential getSignedCredentials(in string user, in string password,
in StringList roles);
SignedCredential getSignedCredentialsGivenKey(in string ASid,
in SignedCredential encryptedStr);
SignedCredential getRestrictedSignedCredentials(in SignedCredential cred,
in StringList restrictions);
SignedCredential getNonDelegatableSignedCredentials(in SignedCredential cred,
in string targetService);

SignedCredential getSignedCredentialsAddRoles(in SignedCredential cred,
in StringList roles);
SignedCredential getSignedCredentialsDeleteRoles(in SignedCredential cred,
in StringList roles);

string  getUserIdentity(in string majorId, in string minorId);

// Administrative Functions

void addDeviceUserMapping (in string majorId, in string minorId, in string userId);
void addUserPasswordMapping (in string userId, in string password);

};
```

Figure 4: CORBA IDL for Authentication Service

getNonDelegatableCredentials takes a credential and returns another which is non-delegatable. **getCredentialsAddRoles** and **getCredentialsDeleteRoles** take care of adding and deleting roles from the credential.

The Authentication Service has a mapping similar to the `/etc/passwd` in UNIX which forms its database. In addition to user, password it also contains a list of the roles and attributes which the user possesses.

```
user   encrypted-password  attribute-list role-list
```

Figure 5: Structure of the Authentication Service Database

2.5 Implementation of Credentials

Credentials will have the structure as shown in Figures 1,2,3. These are signed by the Authentication Service using its private key. The Access Control Service which uses the credentials can decrypt it using the public key of the Authentication Service. GSS-API is used for managing keys and performing encryption and decryption. The internal structure would probably be represented using XML.

3 Access Control

Access Control in GAIA is discussed in details in [1]. Here we provide a very short description of the same.
GAIA will provide an extensive mechanism to enforce different kinds of access control. It provides:

1. Role-Based Access Control.
2. Discretionary Access Control.
3. Mandatory Access Control.

Role-Based Access Control is applied to all the System Files and Services. **Discretionary Access Control** is provided to users so that they can secure their private files. **Mandatary Access Control** provides military grade security in the system.

3.1 Interface for Access Control Service

The interface for the Access Control Service is shown in Fig 6. The Access Control Service on obtaining the name for the Resource can fetch the corresponding policy file and decide whether or not access is allowed. The two methods available in the interface for the Access Control Service are **canAccessGivenPolicy** and **canAccessGivenPath** both of which

```
#include "tao/IOP.pidl"
interface AccessControl {

        typedef sequence<octet> Credential;

        typedef sequence<octet> Literal;

        typedef sequence<Literal> Term;

        struct MethodPolicy {
                string methodName;
                sequence<Term> terms;
        };

        typedef sequence<MethodPolicy> MethodPolicyList;

        typedef sequence<octet> Argument;
        typedef sequence<Argument> Arguments;

        boolean canAccessGivenPolicy (in IOP::IOR svcIOR, in Credential cred,
                                      in MethodPolicyList mpList, in Arguments
args);

        boolean canAccessGivenPath (in IOP::IOR svcIOR, in Credential cred,
                                      in string accessPath, in Arguments
args);
};
```

Figure 6: CORBA IDL for Access Control Service

take a Credential as input. They also take Arguments as a parameter as some policy might be specified in terms of the arguments. The svcIOR parameter is needed in the case of non-delegatable credential. The mpList and accessPath parameters are used by the access control service to access the policy.

For example, when a user wants to access a file, he uses the File Service. The File Service contacts the Access Control Service with the path for the File and the credentials of the user. The Access Control Service then tells the File Service whether the user is authorised or not to access the file. The credential is generated by the Authentication Service and is encrypted. The Authentication Service and Access Control Service share the same key.

The Access Control Service makes its decisions based on the policy file for the resource/object/service. These policy files can be edited by the owner of the resource to give permission to others..etc. There are also meta-data files which determine who has access to the policy files.

4 Secure Dynamic Loading of Components

There are several aspects to this problem which are discussed in the following subsections.

4.1 Component Repository as part of the file system

The component repository can be stored on the file system. This simplifies its security management and further ensures that the generic security mecahnism for the system can be used for this too.

4.2 Access Control for Component Repository

There are two sides to access control for the component repository. First of all, a policy dictates who can place components into the component repository. For example, if components are stored in a directory called `/ComponentRepository` the policy for the directory can specify who all can add their components to the Component Repository. A user can be given permissions to add components to the Component Repository by giving him write permissions for the directory.

Secondly, when a user adds a component to the Component Repository , being the owner of the component(file) he can create a policy for it. Using this policy he can limit access to the component to certain users or roles or users satisfying certain attributes.

4.3 Integrity of Components

A user might not want to run components which are not signed by a trusted authority. Therefore components could be signed to verify its authenticicy. A component can be signed by a signing authority and verified prior to loading it.

4.4 Access Control in the Component Manager

Also, it **might** be necessary to allow different users to instantiate components inside a component container. This is done by associating security policies with each component. A Secure

Component Manager provides such a functionality. Each component is started with a policy file. The creator of the component specifies access to certain users or roles using this policy file.

4.4.1 Usage

The **-p** option is used to specify a policy file for a component while creating it. For example, the commandline shown in Fig 7, it starts a ComponentContainer with a CORBAComponent-ManagerExporter associated with a policy file **cc.pol**.

```
ComponentContainer SYSTEM -cBase/InterceptedTAOExporter
-cBase/CORBAComponentManager -cCORBA/CORBAComponentManager -pcc.pol
```

Figure 7: Creating a component with a policy

```
createComponent
Junior_Admin
;
```

Figure 8: Example of a simple policy file

If **cc.pol** has an entry such as the one shown in Fig 8 it a client is allowed to create a component only upon presenting the credential of a JuniorAdmin. Thus policies can be specified for each method. In the case of the CORBAComponentManagerExporter we can allow creation of a component only upon presentation of a certain credential. Such policies can be specified for any component.

However, such Access Control is possible only for components which have been *exported* to the outside world. Within a ComponentContainer it is not possible to isolate one component from another as they reside in the same address space. Also, InterceptedTAOExporter is required instead of TAOExporter to make the security mechanism work.

4.4.2 Credentials in the Component Container

This section explains how credentials are associated with components and provides some insight into the implementation.

With every component in the ComponentContainer two credentials are associated: **"self credential"** and **"client credential"** (See Fig 9. Consider a component container CC. If user A creates and loads a component inside the component container user A's credential is the component's **self credential**. Now a client C may make request invocations on this component. In such a scenario C's credentials are assigned to the **client credential** of the component. Thus the component possesses both the creator's and the client's credentials and can use them appropriately. For example, when the component wants to perform certain operations on behalf of the client it shall use **"client credential"** and when it wants to use its own credential it shall

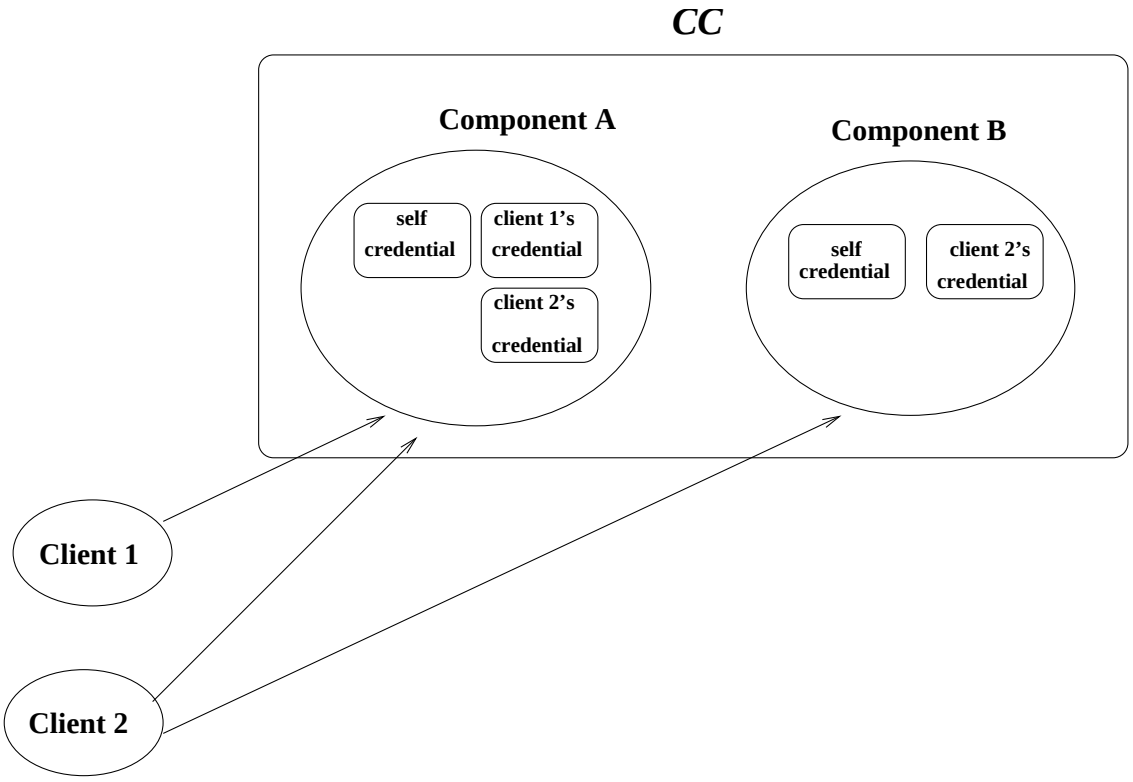

Figure 9: Credentials in the ComponentContainer

use **"self credential"**. The client's credentials are obtained using Interceptors.

On creation the **"self credential"** of the component is set to that of its creator. When a component serves a client the interceptor transparently extracts the credential of the client's credential and sets the **"client credential"** of the component. In case of more than one client making requests on the component concurrently a mapping of Client thread's and credentials is maintained.

The interceptor on the client side extracts the **"self credential"** and passes it transparently along with the request. The interceptor on the server side extracts this credential and sets the **"client credential "** on the server component. If the client wishes to use any other credential other than the **"self credential"** it sets the **"self credential"** approrpiately. The Secure Component Manager has the following methods which can be called to acheive this objective. Please see Fig 10.

Note: getSelfCredential will be changed so that it doesnt require a thread id to be passed to it.

5 Secure Boot-strapping and Login

The bootstrapping is performed across two levels. The first boot level consists of boot-strapping the Security Services, Naming Service and other System services like the Trader Service (top level). These constitute the Trusted Computing Base for GAIA.

```
SignedCredential& getSelfCredential(int id);
SignedCredential& getSelfCredential(const char* serviceName);
SignedCredential& getClientCredential(int id);
void setSelfCredential(int id, SignedCredential& cred);
void setClientCredential(int id, SignedCredential& cred);
```

Figure 10: Interface to set and get credentials

At the second level, an active space is boot-strapped. This includes loading a Boot-strap component and the Login Component. To ensure the integrity of the boot process, the second level verifies the integrity of the first level. This process and the overall boot-strapping process is explained in more detail in another document[2]. Here we just present an overview of the whole procedure.

The Nameservice is modified so that it can take a configuration file which contains the IOR's of the security services, Trader..etc and the private key of the GAIA-Admin (which is the only role allowed to boot the system). It also encrypts its own IOR using the GAIA-Admin's private key and binds it to a context. When the second boot level starts, the Component container verifies the authenticiy of the Naming Service by decoding the encrypted IOR with the public key of the GAIA-Admin and checking it with the one advertised as the IOR of the Naming Service.

The login component in GAIA takes in a login and password and contacts the authentication service to get a credential. It then contacts the Environment Service to retrieve a set of last known list of roles in the user's environment and asks him to to select a list of roles he wants to use for the session. It then gets a credential from the authentication service which includes the set of roles chosen by the user. The authentication service grants the user the set of roles asked for after verifying the role hierarchy of the system.

6 Need for a "SetUserID Service"

As of now a component inherits the same credential as that of the creator. It is possible to make a credential distinct from that of its creater by simply adding another field to the credential which indicates the component name. This way when a component behaves maliciously it will be possible to pinpoint the blame on it. However in some cases a component may want to create another component but endow it with a credential which is more powerful than its own. For example, a janitor should be able to start an active space (if the active space's policy allows it) but then the active space must run with its own credential. In such a case a service called the SetUserID service starts the Active Space object with the desired credential. A component requiring to start a component in such a manner with a promoted credential should contact the SetUserID service which accomplishes such a task.

The SetUserId service looks at system policies which dictate what components can be started with promoted credentials upon. This interaction is shown in Fig 11.

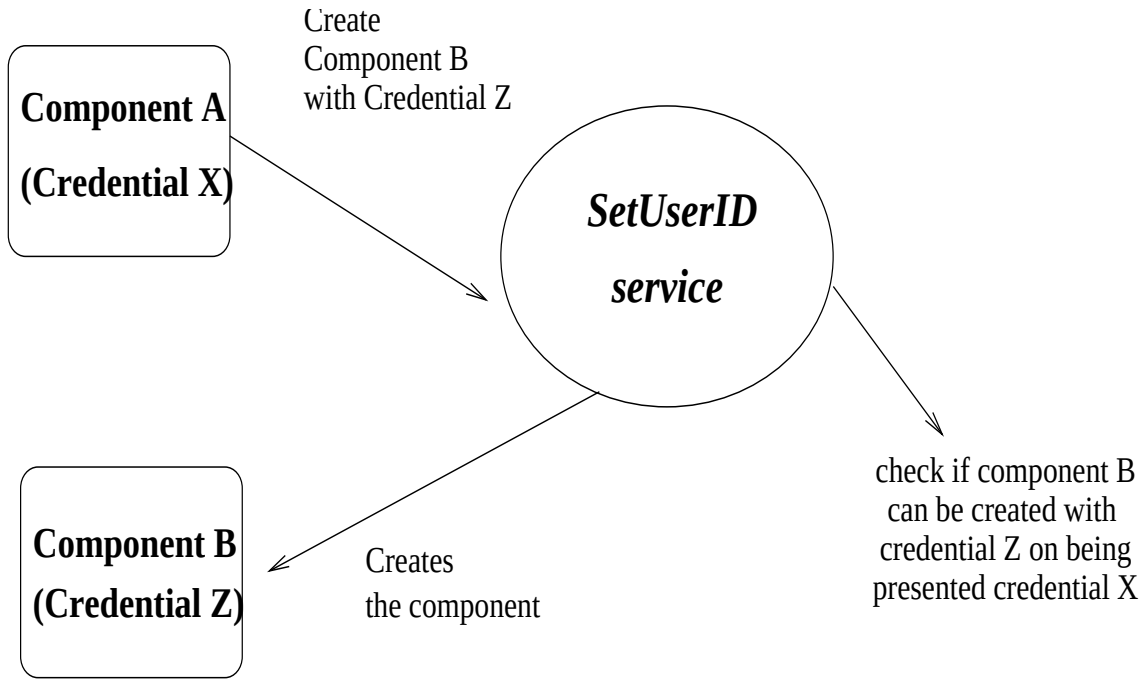

Figure 11: SetUserID service

7 Secure Tracking and Location Privacy

Secure Tracking gives a user of the system privacy. A user can decide not to be tracked and monitored in the Location Service. *more on this later*

8 Architecture

The Security Services in GAIA consist of

- Access Control Service (See 3.1).
- Authentication Service (See 2.4).

In addition the Component Container started for bootstrapping an active space has a minimal "crypto" library. This is necessary to verify the integrity of the Name Service.

References

[1] Binny Gill. "Access Control in GAIA".

[2] "Secure Boot-strapping".