Generation of Role Based Access Control Security Policies for Java Collaborative Applications

J. Briffaut – X. Kauffmann-Tourkestansky
J.-F. Lalande – W. W. Smari

LIFO
Université d'Orléans / ENSI de Bourges
France

Electrical and Computer Engineering Department
University of Dayton
USA

SECURWARE'09

Access control for software

- DAC : Discretionary Access Control
  - Permissions defined by Owner
- MAC : Mandatory Access Control
  - Permissions defined by Administrator (Independent User)

- Controls a software, a database, etc...
- For a Java software classicaly:
  - The operating system enforce a DAC policy
  - Extra tools provide a Mandatory Access Control Mechanism
- RBAC: Role based Access Control

MAC mechanism for Java software

Two running modes:

- All permissions are granted
- The software is sandboxed

Sandboxed software:

- Network is limited
- Read and Write permission are very limited
- Graphical operations can be forbidden

Limitations:

- On what URL ?
- On what system object ?

JAAS permissions

Class control:

```
permission java.lang.RuntimePermission "accessClassInPackage.sdo
      .foo";
```

I/O control:

```
permission java.io.FilePermission "tmpFoo", "write";
permission java.io.FilePermission "<<ALL FILES>>", "read,write,
      delete,execute";
permission java.io.FilePermission "${user.home}/−", "read";
```

Network control:

```
permission java.net.SocketPermission "∗.ensi−bourges.fr:1−", "
      accept,listen,connect,resolve"
```

JAAS example policy file

Example of policy file: *~/.java.policy*

```
keystore "${user.home}${/}.keystore";

grant codeBase "file:${java.home}/lib/ext/−" {
  permission java.security.AllPermission;
};

grant codeBase "http://www.ensi−bourges.fr/files/" {
  permission java.io.FilePermission "/tmp", "read";
  permission java.lang.RuntimePermission "queuePrintJob";
};
```

- The user has to write the policy
- He cannot be helped by the developer

Objectives

To give to developpers a solution that:

- Provides a way to define the policy in the code
- Introduces roles in collaborative software
- Gives an RBAC API for the software

The users will be able to:

- Collect needed permission and take a decision
- Choose a role in the software

Security:

- Is this sufficient to control a possible vulnerability in the software ?

Language

Inspired from SELinux rules:

```
allow <subject> <IT> <object>
```

Javadoc comment before methods:

```
/**
* @allowIT Root {all}
* @allowIT User{awt} "accessClipboard"
* @allowIT User{file:(read);file:(write)} "config.txt" */
```

Language deployment

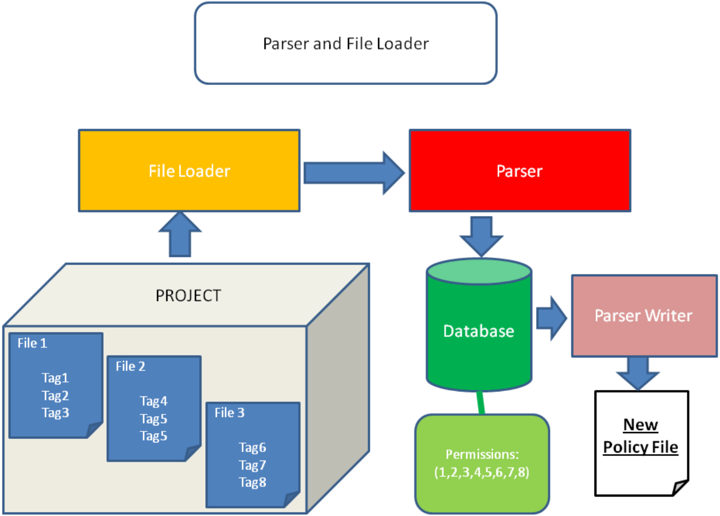
A parsed rule in the Java code:

```
/**
* @allowIT Root {file:(read);file:(write)} "config.txt" */
public void convertConfiguration()
...
/*
* @allowIT Root {file:(read);} "password.txt" */
public void authenticate(String password)
...
```

will produce:

```
grant Principal test.JAAS.ExamplePrincipal "Root" {
permission java.io.FilePermission "config.txt", "read";
permission java.io.FilePermission "config.txt", "write";
permission java.io.FilePermission "password.txt", "read";};
```

Tag loading and policy deployment



Login module

Login module provides to the software:

- Hypothesis: the authentication is done
- Called at any time of the software
- Proposes to a user to obtain a role

The login module then checks:

- That the user can take this role
- That the right policy is loaded in JAAS

Login module

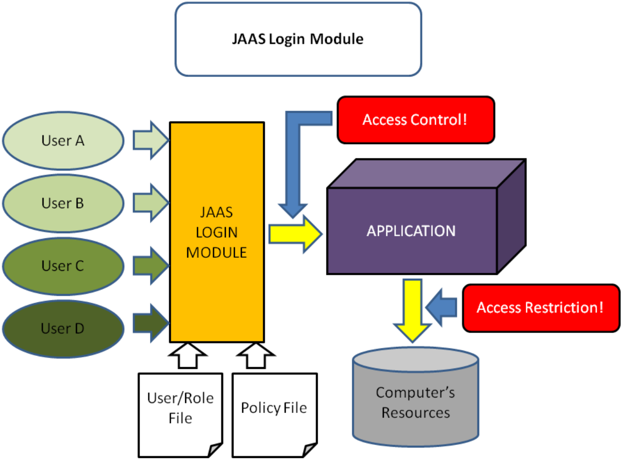


Figure: Login module

Benefits for collaborative applications

Benefits of this architecture:

- It eases the security policy generation
- It allows sandboxing the application
- It adds an authentication security level before using the application
- It simplifies the writing of policies for developers

The design of the policy is

- Collaborative for developers
- Controlled user by user by the administrator
- Gives guarantees for users

Collaborative design of the policy

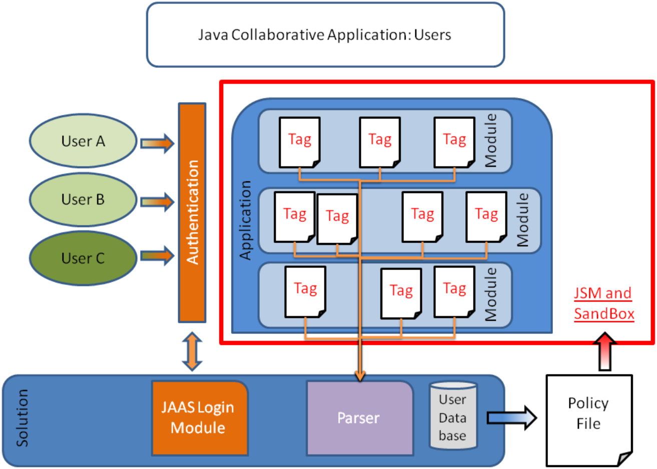


Figure: Java collaborative application

Collaborative design of the policy

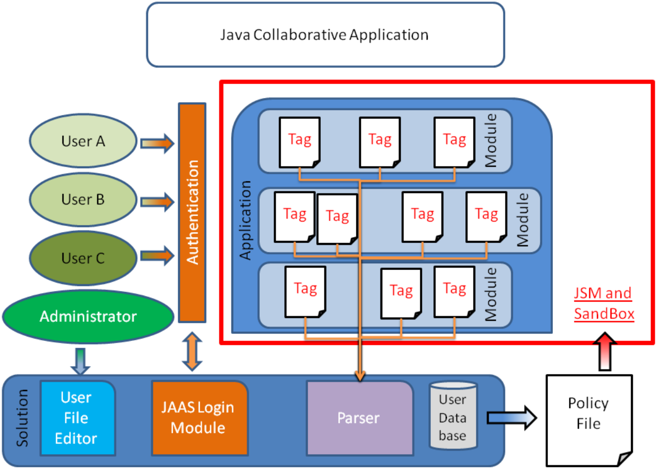


Figure: Java collaborative application

Policy tags

```
/**
* @allowIT Root {file:(read);file:(write)} "config.txt"
* @allowIT User {file:(read)} "config.txt" */
public void convertConfiguration()
...
```

generates...

```
grant Principal test.JAAS.ExamplePrincipal "Root" {
permission java.io.FilePermission "config.txt", "read";
permission java.io.FilePermission "config.txt", "write";
};
grant Principal test.JAAS.ExamplePrincipal "User" {
permission java.io.FilePermission "config.txt", "read";
};
...
```

With the Root role

The user choose the Root role:

Listing 1: Console output for a read+write operation

```
Password linked to the username in the userfile:
a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
Hash of the inputed password :
a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
Authentication succeeded!!

Please pick a role:
---- >Root
Role pick succeeded.
Actions done once authenticated ..........
The file exists in the current working directory
the file has been read!
The file config.txt was created (write test)!
```

With the User role

The user choose the User role:

Listing 2: Console output for a read+write operation

```
Password linked to the username in the userfile :
a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
Hash of the inputed password :
a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
Authentication succeeded!!

Please pick a role :
−−−− >User
Role pick succeeded.

Actions done once authenticated ..........
The file exists in the current working directory
the file has been read!
java.security.AccessControlException: access denied
(java.io.FilePermission config.txt write)
```

What about vulnerabilities ?

What about a software vulnerability ?

- Hypothesis: an attack succeeds against the software
- If the choosen role is root...
- ... the attacker will be able to write in config.txt !

For example:

- A Peer-to-peer application with a network vulnerability
- A web server application on a Tomcat platform

The permissions are 99% of the time useless...

Permissions are useless ?

```
/**
* @allowIT Root {file:(read);file:(write)} "config.txt" */
public void convertConfiguration()
...
/*
* @allowIT Root {file:(read);} "password.txt" */
public void authenticate(String password)
...
```

These permissions are useless:

- Permissions on config.txt are useless in *authenticate()*
- Permissions on password.txt are useless in *convertConfiguration()*

Hypothetical vulnerability

```
/**
* @allowIT Root {file:(read);file:(write)} "config.txt" */
public void convertConfiguration()
...
// Vulnerability at this point: injecting this code:
   this.passwordFileObject.println("hacked password");
...
/*
* @allowIT Root {file:(read);} "password.txt" */
public void authenticate(String password)
...
```

- Arbitrary access is allowed to password.txt
- Even if multi-threaded, the code have no reason to have permanent access to password.txt

Proposed solution

To dynamically enforce the policy when required:

```
/**
* @allowIT Root {file:(read);file:(write)} "config.txt" */
public void convertConfiguration() {
RBAC.loadPolicy("root_convertConfiguration");
...
// Vulnerability at this point: insecting this code:
   this.passwordFileObject.println("hacked password"); // This will fail !
...
RBAC.unloadPolicy("root_convertConfiguration");
}
/*
* @allowIT Root {file:(read);} "password.txt" */
public void authenticate(String password) {
RBAC.loadPolicy("root_authenticate");
...
RBAC.unloadPolicy("oot_authenticate");
}
```

Conclusion and perspectives

The implemented RBAC module proposes:

- A tag parser and policy generator
- A login module for software integration
- A dynamic method of policy enforcement

What next ?

- Extract automatically policies from source code
- Link JAAS to SELinux ?

Questions

- Questions ?