Final Project Report for CS 557

Yunhe Tang, Jun Tang, Di Wang

Introduction:

In this project, we built a distributed role-based access control system. Access to resources is regulated by the process of authentication and authorization. Authorization is handled by multiple agents. They would issue a client certificates to prove their role in the system, according to their local knowledge, access control policies and certificates issued by other agents it trusts. Each agent could choose to trust some other agents or not. Finally, the authorization to access some resource is determined by a centralized Service agent which would choose to authorize or not the client in a similar way as mentioned above.

Keywords: Role-based; Trust Management; Access Control; Prolog

System Overview:

In our system, from the perspective of users there are mainly two kinds of resource--*documents* and *code*, and three kinds of role -- engineer, manager and the generic employee. *Document* is available to all employees of this company. Code can be edited only by *engineers* but can be read by both *engineers* and managers. Currently both document and code are just dummies in the form of key-value pair.

In addition we have 3 authorization *Agent*: one is a department HR, who's responsible for confirming the employment of some employee; also we have a company HR, who's responsible for confirming the role of an employee; at last we have Service agent, which controls all files/resources and is the ultimate agent who determines whether an agent would be authorized for some operation. To access resources in the system, a user need to launch a client-side application.

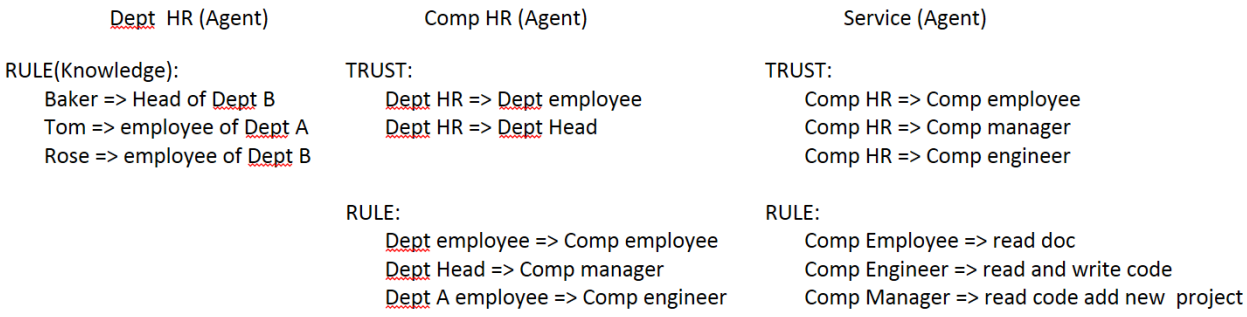

Fig. 1 A Subset of Prolog Rules (Dept for Department, Comp for Company)

A typical use case of our system could be engineer Tom wants to read the design code of an ongoing project. After he launches the client-side software with his password, he types in "readDoc,#" (to request a reading operation). The client-side software would first talk to the department HR and get "signed certificates" to confirm he's from the engineering department (A.K.A. department A in our design). Then the client-side application would talk to

company HR and get a certificate stating that he's an engineer of this company. Finally, Service would be contacted, and all certificates a client have would be presented. The Service agent would then initiate the prolog engine with the received certificates and local policies. As the Service's local policies states that engineers can read code, and Tom has a certificate from company HR, which Service trusts, proving him to be an engineer, Tom would be granted to read the code.

Security goal:

The first goal of our system is to regulate the access to resources in the system. In addition, we would like to to accommodate the fact that usually information about people's identity and policies about authorization is distributed among multiple departments/agencies.

Adversary Profile:

We assume the most luring goal for adversaries is identity theft: to steal the identity of some legitimate user (possibly some high rank manager) and get access to some sensitive information or modify some important files. Another goal of them maybe to impose as some agent in the system, whether to corrupt the system's functionality or help some fake users to steal identity. Finally some adversaries may want to pretend to be the Service -- the agent controlling all resources. This is a potential way to inject fake data into the system when clients are requesting data from service; or it can be a form of leakage when user are trying to update some files with new data.

We assume the adversary would be able to contact our agent from outside of our system. We also assume a legitimate user (employee) in our system may try to some malicious operations. However, we assume adversaries would not be able to corrupt or steal private keys of other users or Agents in our system. We also assume they are not able to corrupt our prolog files as access to them would be restricted by OS.

System Design:

1) distributed authentication: we have implemented a distributed authentication system, for the purpose of ease of use. For example, company HR department is responsible for recognizing who is engineer and who is manager. while a department HR can only decide who is from which department. Because in reality, a company has multiple departments, each departments is in charge of managing their local policy. In this base, the security of overall system is strengthened. Because even if we have a malicious user, who is a employee of a company, he is limited to change and break the policy that he is in charge of. In other words, other policy or data is unreachable to him. This brings additional robustness to the overall system. Because if one department crashed, as long as other server works well, the whole transaction model still holds. In short, the distributed manner restricts the scale of effect that sub function failure could cause.
2) why prolog: flexibility, easier to modify policy dynamically--whether delete or change or add new.prolog language is a logic language, which can make logic deduction based on the facts and rules stored in its own knowledge database. We take advantage of

prolog to implement the authorization mechanism, such like who can read the code or who can write the code. To a large extent, prolog helps us to improve the flexibility of rule making. Suppose that during the use of the project, we want to add new rule dynamically, we can add it to our database file directly, instead of changing our code. Prolog also allows us to assert temporary statement into its knowledge database dynamically. For example, if we want to make a deduction based on additional fact that is not included inside the knowledge database, we can insert the fact into database temporarily. After the execution of prolog, we delete the inserted fact. As if the fact never appears.

3) config file to make easier, path of key file easier for testing. we set up a configure file to for client program so that the client could change the launch parameter conveniently.
4) Certificates are formatted in this way: it starts with a “SEP” to denote the starting of the certificate, then 3 digits to specify the length of the statement/content of the certificate. The first five letters in the content is the name of the agent issuing the certificate, after that is the content of the certificate. Following that, is the ciphertext generated from the hash value of the content by encrypting with the private key of the signing agent. A actual certificate states that baker is an employee of company is shown in Fig. 2.

SEP026COMHRcomEmployee(baker)aK4n4wo3ZfA9zNxcZwK2EODcFTXKMpPsBTqGkZ8PLx+9odyVR
wcFd5ycZHoi2FtN7RW7Orf8DMAy
Gxt+hansBr/PIflVhQdb6XMfaTpHSgjgiAIsE7siXCDd/hGI4PbdIUFc8sJ4IckygOrnvbMHwF3I
RFkyZZAWy813nf1NXp7WWYT0COMRJL/MudNM0M96x4QTVFM8H2VC4kDYE+UsTgi6qVuNg7aOhM1j
dCiqWZ4rMaKOEpxAeCW56h0q4aco0UrS12CHLA1vO2canZcDGZ2hBvBhg/+x0M8mCeJ4aCReIzRN

Fig. 2 Sample certificate from Company HR stating that baker is a employee

5) To verify and extract information from a certificate, recipient of a certificate would first check the name of the agent who issued this certificate by checking the five letters after the length of the content. Second, the recipient would try to use the public key of the issuer to decrypt the ciphertext. Third, the recipient would hash the content and match it with the deciphered value. If they pass the match, the certificate is accepted as a genuine certificate, the content of the certificate is added into memory as a fact and used for this one time.
6) SSL is used for all connections
7) Password is not shown in terminal when input. whenever a user is inputing a password, the text content of the password is not shown on the terminal. it is designed this way to protect from shuffle attack.

Protection:

Identity theft:

The password used for generating the certificate is used for logging in. In addition, when typing in, the password is “hidden” in a Linux/Unix manner. For any adversary who want

to attack this system, he/she must get the certificate of the some ligitimate client and his/her password.

Imposter:

Some adversary may be interested in impost as some agent or even the Service agent, this is a possible way to inject fake data into the system when clients are requesting data from service; or it can be a form of leakage when user are trying to update some files with new data. This attack is prevented by SSL connection and the.

Denial of Service:

Theoretically, our current implementation is vulnerable to denial of service attack. But because of limitation of time, we didn't implement any defense mechanism in this aspect. In addition we think it's reasonably to assume this system would be used in a local area network of some company, the risk to encounter a denial of service attack is very low.

Trespassing:

Some legitimate users of the system may try to access some files which they shouldn't by intention or failure/ignorance. This kind of attacks/faults would be denied because the deduction of prolog would not authorize them.

Flaws and Future Work:

An important flaw in our current implementation is that the prolog file which forms the basis for our authentication/authorization deduction is "naked". An intuitive way to solve this is to protect the prolog file with encryption. Unfortunately, the jpl java interface of prolog only supports reading prolog statements from a plaintext file, and we haven't find anyway to eschew that. Some possible ways to mitigate this problem include: 1) use temporary files--store prolog file permanently as a ciphertext, decipher it and store it in an temporary file for operations and delete it after that; 2) limit the access to prolog file--e.g. require root privilege.

To prevent a user from using expired certificates in a "replay" manner. One intuitive solution is to add a field in the certificate to specify the expiration date (and time) of that certificate, and every agent in the system would only accept non-expired certificates as valid. Another solution maybe require the user to contact the service before each operation and get some "random number" -- some specific professiona term? -- and the user can pass it on to every agent he/she contacts. In turn all agents would include that random number in the certificates, so that when the service receives those certificates, it could be sure all those certificates are fresh and dedicated for this operation.

One interesting direction to explore would be adding an agent who handles all procedural-information. This may solve one of the inconvenience of our system: the procedure to get certificates from different agents are hardcoded in the program, and the certificate agent just gives a client all the certificates he/she is eligible to have without any discrimination whether they would be needed. One possible way to mitigate this is using some agent as a reference (e.g. Ref-Agent). Every time a client wants to do something, he/she contacts this Ref-Agent and ask which agents he/she should contact and which certificates he/she would need, this response could also come as a signed certificate. So the client could

contact the appropriate agents; on the other hand, agents may choose to only give a client the certificates he/she needs for the requested operation instead all that is eligible for the user.

Another interesting direction to explore maybe to make the system more “distributed” by distribute the responsibility for authorization. Currently, the authentication is distributed into multiple agents, while the *Service* is actually the only agent who determines whether an operation would be authorized. This may provide too much burden on the Service agent and it’s not very realistic. One possible way to improve this may be use other agents to do authorization too. For example, public relation department’s HRs may be given the instructions that employees of their department are not allowed to read any source code of the engineering department. As a result, when Tom ask them to sign a certificate to prove he’s an employee of their department so he could contact Service to read some code, the HR of PR may just deny his request.

Alternative Approaches:

It seems practical to use different formats for certificates. For example, right now department HR and company HR send out certificates one by one, each containing only one statement. An alternative way is to combine all statements in one certificate. This may save some computation in turns of hashing and encryption, but it would also limit the user’s freedom to use these certificates separately.

A similar system may also have a very different way for distributing and presenting certificates. Currently, the authentication agents in our system would give the client all the certificates he/she is eligible to get; on the other hand, client-side application would present Service with all the certificates it has. In both cases, some of the certificates may not be needed. So an attractive alternative approach is to store some information about which certificates would be needed for which request explicitly in the system. However, this is beyond the intended scope of this project, especially so when considering the potential problem that this information/policies may not be synchronized properly across multiple agents.

Conclusion:

In this project, we have developed a distributed access control system. At the base of our system is the Prolog engine, which does deduction according to local rules and knowledge to determine which certificates should be issued (or which access should be granted). With certificates and the trust among agents, policies/knowledge distributed in the system would be essentially integrated to determine whether a client could have access to some resource. In addition, all clients are evaluated by their role in the system.

Using prolog made it possible to modify access control policies in the system dynamically without changing the implementation of the software. Adapting the role-based approach makes our system more flexible -- it resembles the concept of “group” in some other fields. Using “trust” as a way to propagate authentication/authorization information makes our system modular and more resilient to corruption of part of the system.