Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition

061302
Introduction

• **Securing availability of services is challenging**
  - The telecommunications environment has evolved (from centralized to distributed)
  - Services are being developed in a distributed manner in a connectionless environment requiring cooperation of several components and actors
  - Security requirements are not taken into account in the design process
    - *Time to market and cost constraints*
    - *Lack of knowledge about security*
    - *Complexity of environment in which services are deployed*

• **Service Composition - allows for incremental service development.**
  - service components are dynamically combined at run time
  - Allows for reuse of components

• **Our approach:**
  - Development of (re-usable, flexible) patterns to ensure availability in composition
Service Oriented Architecture

- Cross-cutting nature of services:
  - Service components interact with each other for the execution of services
  - Dynamic linking - fundamental and general mechanism in S-O systems
    - Creation and release of dynamic links: mechanisms for service discovery, feature selection, compatibility validation, and access control
Availiability Decomposed

• ISO/IEC TR 13335:
  “Availability is the property of being accessible and usable upon demand by an authorised entity”

• Availability is a composite notion consisting of:
  • **Exclusivity** - the ability to ensure access for authorised users only
  • **Accessibility** - the property of being at hand and useable when needed
Analyzing availability from different viewpoints

Viewpoint of the user

Black box view

Service viewpoint

System viewpoint

White box view

Service components + Service availability components

System components + System availability components
AA patterns Framework:

- We present a framework and classification of authentication and authorisation patterns
  - For composing with services
  - To ensure that services are accessible to the authorised users only.
- We demonstrate how the authentication and authorisation patterns can be composed with services to ensure that access to services is granted to authorised users only. This involves
  - Employing policies (role binding policies) to specify rules
    - Compatibility of roles
    - Is actor/agent authorised to play a role
    - Is playing of a role allowed by the actor/agent
    - Policies are specified in OCL.
  - Using goals, and/or pre- / post- conditions
  - Employing Semantic Interfaces to define the interface behaviour between participating roles
Classification of authentication patterns

- TwoParty Authenticate
  - Unilateral Authenticate
    - UniOnePass Authenticate
    - UniTwoPass Authenticate
  - Mutual Authenticate
    - MTwoPass Authenticate
    - MThreePass Authenticate
Unilateral one pass authentication patterns

- UniOnePass Authenticate
  - UniOnePass Authenticate Symmetric
  - UniOnePass Authenticate Asymmetric
  - UniOnePass Authenticate Hash Function
  - UniOnePass Authenticate crypto check function
Classification of authentication patterns

- **Rationale:**
  - Describe generic patterns first
  - Separate the choices that the developer must take
  - Pinpoint each of the levels of specialisation for re-usability, flexibility and awareness
    - Protocol
    - Algorithm
    - Key size

**Why?**
- Flaws may be introduced in each of the layers of specialisation

**How?**
- Specify the (policies) properties/requirements on the instances playing the roles independently of choice of protocol/algorithm
- Employ UML 2.0 Collaboration uses and Interaction uses to facilitate re-usability of the patterns
Specifying AA-patterns

- UML 2.0 collaboration diagram for generic two party authentication pattern

\[
\text{TwoPtyAuthenticate}
\]

\[
\text{authenticatee} : \text{aType} \quad \text{ authenticator} : \text{bType}
\]
Specialization - view 1

- UML 2.0 collaboration diagram for unilateral two pass authentication pattern
  - Goal expressed in OCL

```
{def: goal : Boolean = authenticatee.Unilaterally_Authenticated}
```

```
authenticatee : responder
```
```
authenticator : challenger
```
Specialization - view 2

```
{ Context c: UniTwoPassAuthenticate
  Inv:
  c.authenticatee.GenerateResponse.is_generatable AND
  c.authenticator.GenerateChallenge.is_generatable AND
  c.authenticator.ValidateResponse.is_validatable
  Pre:
  c.authenticatee.secret.is_assigned AND
  c.authenticator.knowledge.is_assigned AND
  Relation (c.authenticatee.secret, c.authenticator.knowledge)}
```
Specialization - view 2

```plaintext
UniTwoPass Authenticate

<table>
<thead>
<tr>
<th>responder</th>
<th>challenger</th>
</tr>
</thead>
<tbody>
<tr>
<td>secret : string</td>
<td>challenge : string</td>
</tr>
<tr>
<td>algorithm : string</td>
<td>knowledge : string</td>
</tr>
<tr>
<td>GenerateResponse ()</td>
<td>GenerateChallenge ()</td>
</tr>
<tr>
<td></td>
<td>ValidateResponse ()</td>
</tr>
</tbody>
</table>

authenticatee authenticator

{ Context c:UniTwoPassAuthenticate
  Inv:
  c.authenticatee.GenerateResponse.is_generatable AND
  c.authenticator.GenerateChallenge.is_generatable AND
  c.authenticator.ValidateResponse.is_validatable
  Pre:
  c.authenticatee.secret.is_assigned AND
  c.authenticator.knowledge.is_assigned AND
  Relation (c.authenticatee.secret, c.authenticator.knowledge) }
```
User Pull patterns

User Pull

Access Server

User

Service Access Filter

UAs1: TwoParty Authenticate

authenticator

authenticatee

authenticatee

UAs2: Auths Activation

authenticator

authoriser

authorisee

USaf2: Checking Access Rights

data

User Pull patterns
- Using semantic interfaces:
  - Defined based on role modelling and simple goal expressions
  - Facilitate validation of safety and liveness properties
  - Define visible interface behaviour and goals of the collaboration
  - Declaration of role-binding policy is useful
    - \textit{validation that security properties are preserved in composition of the pattern and services.}
Specification of AA patterns:

- Specification of each pattern as a UML2.0 collaboration with semantic interfaces
  - For which interface behaviour for each of the roles in the collaboration is defined
  - Role-binding policies are annotated
    - *To enable us to validate that the required conditions have been fulfilled in order for*
      - Authentication pattern to run correctly
      - So that availability requirements are fulfilled when composing the pattern with services

- Why?
  - Semantic interfaces facilitate validation of safety and liveness properties
  - Checking compatibility of different components involved in a service collaboration
    - *By validating the interface behaviour instead of the component as a whole*
    - *Policy rules to check:*
      - Compatibility of roles
      - Is actor/agent authorised to play a role
      - Is playing of a role allowed by the actor/agent
Role-binding policies - constrain binding of roles to agents at run time

{ Context c:UniTwoPassAuthenticate
  Inv:
  c.authenticatee.Generate Response.is_generatable
  Pre:
  c.authenticatee.secret.is_assigned
  Post:
  c.authenticatee.playRole }

{ Context c:UniTwoPassAuthenticate
  Inv:
  c.authenticator.GenerateChallenge.is_generatable AND
  c.authenticator.ValidateResponse.is_validatable
  Pre:
  c.authenticator.knowledge.is_assigned
  Post:
  c.authenticator.playRole }
Role - binding policy

- With condition on support of the SHA-1 algorithm

```plaintext
{ Context c:UniTwoPassAuthenticate
  Inv: c.authenticatee.Generate Response.is_generatableable
  Pre: c.authenticatee.secret.is_assigned
  Post: c.authenticatee.algorithm = SHA1
  Post: c.authenticatee.playRole }
```
Collaboration policy

- Express constraints that must hold for a collaboration as a whole when it is instantiated.
  - Aim at preventing actions that may compromise the intentions and goals of the collaboration.

```{ Context c:UniTwoPassAuthenticate
  Pre:
  Relation (c.authenticatee.secret, c.authenticator.knowledge)
  Post:
  self.instantiate }
```
Service Composition – Example

- **Service S** defined as semantic interface with roles r1 and r2:
- **Collaboration S** may have a policy P3 specifying:
  - r1 and r2 cannot be played by the same agent
  - Restrictions on types of agents that can play the roles
    - e.g. user agents for r1,
    - terminal agent for r2
Service Composition – Example

- Agents A and B may specify conditions:
  - Agent B specifies preconditions for invoking r2
    - *Pre-cond: A is authenticated and authorised*
  - Agent A specifies preconditions for invoking r1
    - *B is authenticated and authorised*
  - We are looking at using OCL to express these conditions
- Agents A and B may negotiate on the AA patterns to apply
  - E.g., B selects patterns and returns the decision to A
AA behaviour may be invoked in two different situations:

- **Creating a new session by performing a role request - Dynamic role binding**
  - Devise mechanisms to ensure that the role is invoked only if AA policies are satisfied
    - E.g. AA behaviour performed first and desired goal reached before the service is invoked
    - → AA goal is a precondition for service invocation

- **During session behaviour**
  - Required when a session and its service roles contain features or access objects requiring dynamic authorisation (modelled using service access filters, and policies e.g. restricting role behaviour)
  - Required to force termination of a session if authorisations are no longer valid (currently modelled as the interrupt collaboration)
  - Challenging – requires tighter integration of service behaviour and AA behaviour
Service S decomposed

- Binding roles to agents in service composition
- Agent A requests a session of Service S, and role r2 from agent B
Access Server

User

Service Access Filter

Service

{ def: goal: Boolean = User.AAaccessTo(Service) }

UAs1: UnTwoPass Authenticate

authenticator

authenticatee

UAs2: Auths Activation

USaf1: Request Service Access

service access requestor

service access requestee

USaf2: Checking Access Rights

proxied service user

proxy service provider

USaf3: AA Service Use

service provider

US2: ServiceUse

service granter

US1: Request Service

service provider

P1

P2

P3

P4

P5

P6

P7

P8
Authentication and authorisation

- **Authentication** is the process of determining who you are.
  - Authentication binds an identity to a subject. Basis of ID:
    - *Something the entity knows* (e.g. password, PIN)
    - *Something the entity has* (e.g. smartcard, SIM card)
    - *Something inherent to the entity* (e.g. fingerprint, retinal characteristics)

- **Authorisation** is the process of determining what you are allowed to do

- Authentication and authorisation patterns may be used to ensure access to a service to authorised users only.
Elaborated User Pull AA architecture

Overall goal: Expression: Authenticated and Authorised User is granted access to the service

- **UserPull**
- **Access Server**
- **UAs1: TwoParty Authenticate**
- **UAs2: Auths Activation**
- **USaf1: Request for Service Access**
- **USaf2: Checking Access Rights**
- **USaf3: A_Service Use**
- **AsSaf*: Update Access Rights Status**
- **User**
- **Service Access Filter**
- **Interrupted**
- **Interrupter**
- **Proxy_Service_Provider**
- **Authoriser**
- **Authorisee**
- **Service Access_Requestor**
- **Service Access_Requestee**
- **ProxiedService_user**
- **Authenticator**
- **Auths_Granter**
- **Auths_Requestor**
- **Auths_Notifier**
- **Auths_Notifyee**
- **Authenticatee**

The user is granted access to the service after authentication and authorisation.
Thank you for listening

Questions?