IMT4123 - Assignment 3

Ole André Hauge

November 15, 2021

Task 1

Start by studying the standard, and then explain what the attached policies and requests indicate? Determine what the response should be based on the given policy and request set.

Answer: The attached requests and policies sets were: {Request1.xml, Policy1.xml} and {request2.xml, policy2.xml}. Looking at the first set, I assumed that the response would be *Permit* since the policy required the target to have a subject attribute named *Julius Hibbert*, with a resource attribute named *http://medico.com/record/patient/BartSimpson*. Both of which were correctly included in Request1.xml. For the second set I assumed that the response would be either *Deny* or *Not Applicable* since the request does not supply the *account-status* value.

Task 2

Study the chosen API and develop the simple PEP, PDP.

Answer: I chose the *sunxacml* API as it proved to have fairly good documentation in place which would make the process easier. I downloaded the software from `https://sourceforge.net/projects/sunxacml/`. After looking through the Java software package I realized that it came with a fairly good sample directory, in which I found a .jar file for a PDP engine, including a PEP. The .jar file could be run from the command line with two arguments: *request* and *policy*. Based on the supplied policy(s) it would configure the PDP engine and compare the provided request file(s). I tested the tool with some test files to see if it worked, and I found out that I had to set the *CLASSPATH* variable in the command line while running the application in order to make it run correctly. It was run from the sunxacml-1.2/sample directory.

```
java -cp ../lib/sunxacml.jar:../lib/samples.jar src/SimplePDP.java
request/sensitive.xml policy/*.xml
```

As I prefer to work with Python, I optioned to implement the command-line tool in my small python application which I used for the testing of the policies and requests in this assignment. The program can be found in appendix A.

Caveat: I learned that the XACML API I chose to use did not include all the functions like *string-at-least-on-member-of* or *string-bags*, which could have been used to make more efficient policies.

Task 3

Use the application that you developed in the previous task and test the requests and policies given (two sets, each tested individually). Did you get the same result as the one you determined before? Try your application with at least three test scenarios (from the set of the conformance tests) with policies that have three different combining algorithms.

Answer: I continued testing with the files supplied with the assignment, which are included in the *policy-request* directory in appendix A, to see if it matched what I expected from reading the files myself, and got the following results.

Testing Request1.xml against Policy1.xml:

```
Note: src/SimplePDP.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
<Response>
  <Result ResourceID="http://medico.com/record/patient/BartSimpson">
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
  </Result>
</Response>
```

Testing request2.xml against policy2.xml:

```
Note: src/SimplePDP.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
Oct 28, 2021 8:50:57 AM com.sun.xacml.finder.AttributeFinder findAttribute
INFO: Failed to resolve any values for urn:xacml:2.0:interop:example:resource:account-status
<Response>
  <Result ResourceID="CustomerAccount">
    <Decision>Deny</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
    <Obligations>
      <Obligation ObligationId="urn:xacml:2.0:interop:example:obligation:status-code"
      FulfillOn="Deny">
      </Obligation>
      <Obligation ObligationId="urn:xacml:2.0:interop:example:obligation:decision"
      FulfillOn="Deny">
      </Obligation>
      <Obligation ObligationId="urn:xacml:2.0:interop:example:obligation:status-message"
      FulfillOn="Deny">
      </Obligation>
    </Obligations>
  </Result>
</Response
```

Based on these results I can see that my first assessment was right and that it got a *Permit* decision. My second assessment was also right, but I should have seen that it would return as a *Deny* decision and not a *Not Applicable* response.

I then proceeded to test my implementation of the PDP engine with three test policies and request sets, which I downloaded from the set of the conformance tests at: `https://www.oasis-open.org/committees/xacml/ConformanceTests/ConformanceTests.html#Combining\%20Algorithms`.

The first test was with *case 1*, which used a *deny-overrides* rule combining algorithm. The expected result should be *Permit*. The second test was with *case 9*, which used a *permit-overrides* rule combining algorithm. The expected result should be *Permit*. The third test was with *case 17*, which used a *first-applicable* rule combining algorithm. The expected result should be *Permit*. The tests are listed below.

Test 1 result: Permit: RuleCombiningAlgorithm DenyOverrides

```
<Response>
  <Result ResourceID="http://medico.com/record/patient/BartSimpson">
    <Decision>Permit</Decision>
```

```
      <Status>
        <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
      </Status>
    </Result>
  </Response>
```

Test 2 result: Permit: RuleCombiningAlgorithm PermitOverrides

```
<Response>
  <Result ResourceID="http://medico.com/record/patient/BartSimpson">
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
  </Result>
</Response>
```

Test 3 result: Permit: RuleCombiningAlgorithm FirstApplicable

```
<Response>
  <Result ResourceID="http://medico.com/record/patient/BartSimpson">
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
  </Result>
</Response>
```

The files used for the testing are included in the *test-scenarios* directory in appendix A.

Task 4

You are asked to use the XACML policy specification to specify a policy(s) that represents a BLP policy model and test it with your application. You have to create a request to test against your policy, e.g., a subject with a specific clearance wants to access an object with a specific classification.

Answer: The BLP model was implemented with XACML and tested against my PDP-engine implementation. The *BLP.xml* policy file and *BLP_request.xml* files are available in the *bell-lapadula-model* directory in the appendix A. Further assumptions, including the design of the test implementation, are explained in the policy file's policy description.

Task 5

You are asked to use the XACML policy language to specify a policy(s) that represents a Chinese wall policy model. First, discuss the design of your Chinese Wall policy using ABAC, then develop it using XACML.

Answer: With ABAC, access is controlled based on attributes about the subject making the access request and the object being requested, and environmental conditions. Granular policies can then be established on a combination of these attributes to grant or deny access. For example, we can create an attribute-based policy where access is granted only if the values of the project attribute for both subject and object have an identical match. This single policy would ensure that only users assigned to a project get access to the files for that project. The ability to centrally establish and audit access policies across applications, projects, and users is core to an ABAC system. Thus, simplifying access management and reducing risk due to unauthorized access.

Thinking of the Chinese Wall policy model (CWP), we can see that it can be represented by subjects (s), conflict-of-interest (COI) classes, company datasets (CD), and objects. Each object has two

attributes: the name of the corresponding CD, and the name of the corresponding COI, in addition to its object-id. In this model a subject is not limited by a classification level, but have to adhere to two rules for access:

1. Access is granted if the object is in the same CD as an object already accessed by s, or
2. the object belongs to a different COI class.

By implementing a subject with a history attribute containing its already accessed objects, including the objects COI and CD, and by implementing objects with the above-mentioned object attributes we can use an ABAC approach to create a CWP.

The *CWP.xml* policy file and *CWP_request.xml* request file for the XACML implementation of the CWP can be found in appendix A in the *chinese-wall-model* directory. Testing them is done with the function in the main application. More assumptions, including the test design, are explained in the policy-file's policy description.