A Comparison of the Akenti and PERMIS Authorization Infrastructures

David Chadwick, Sassa Otenko
Information Systems Security Research Group, University of Salford, Salford M5 4WT

Introduction

This paper describes the similarities and differences between the Akenti and PERMIS authorisation infrastructures. It describes their features, ease of use and performance statistics. This report was compiled from: a desk comparison of published documentation, by talking to the authors of both infrastructures, and by building both infrastructures along with a test application. The performance statistics are limited to some extent, in that it was not possible to build multiple arbitrarily complex policies in the time available. Also we did not run Akenti as a stand alone server, since PERMIS has no equivalent capability.

Overview

Akenti [Akenti] is an authorisation infrastructure from the Lawrence Berkeley National Laboratory in the USA. PERMIS is an authorisation infrastructure from the University of Salford, UK, and is a product of the EC funded PrivilEge and Role Management Infrastructure Standards validation (PERMIS) project [Permis]. Both the Akenti and PERMIS Authorisation Infrastructures are trust management infrastructures according to the definition of Blaze [Blaze], and have the 5 components necessary for this, which are:

i) A language for describing `actions', which are operations with security consequences that are to be controlled by the system.
ii) A mechanism for identifying `principals', which are entities that can be authorized to perform actions.
iii) A language for specifying application `policies', which govern the actions that principals are authorized to perform.
iv) A language for specifying `credentials', which allow principals to delegate authorization to other principals.
v) A `compliance checker', which provides a service to applications for determining how an action requested by principals should be handled, given a policy and a set of credentials.

Both infrastructures have similar architectures [Johnston] [Chadwick]. This comprises the compliance checker, called the Akenti server by Akenti [Thompson], and the Access Control Decision Function (ADF) by PERMIS (after the ISO Access Control Framework [ISO]). Both have a gateway controlling user access to resources, called the Resource Gateway by Akenti and the Application Gateway by PERMIS. Both of them write their policies in XML, and store their policies in certificates. Both of them can store their user credentials as certificates in LDAP directories. Hence on the face of it, the Akenti and PERMIS authorisation infrastructures seem to be almost identical.

However at the implementation level Akenti and PERMIS are very different. Akenti is written in C++, Permis in Java. The Akenti compliance checker can be called either via a function call in the gateway or as a standalone server via TCP/IP, whereas the PERMIS compliance checker is invoked as a Java object in the gateway. PERMIS credentials are built according to the latest X.509 standard [X509], whereas Akenti credentials are built in a proprietary format [Akenti]. Akenti requires the user to be PKI enabled and to present an X.509 public key certificate (PKC) at authentication time, whereas PERMIS is authentication agnostic and leaves it up to the application to determine what type of authentication to use. Whilst both PERMIS and Akenti policies are written in XML, their DTDs are very different [Thompson] [IFIP]. Furthermore, Akenti ignores the XML in its policy certificates and only uses the Base64 encoded section at the end (which means that the XML and Base64 sections could be different). PERMIS policies are held in one policy X.509 Attribute Certificate (AC), whereas Akenti policies are hierarchical and distributed between proprietary Policy Certificates and Use-Condition Certificates. Akenti has concentrated on classical access control lists (discretionary access controls) whereas PERMIS has implemented role based access controls. Therefore at a practical level there are a significant number of differences between the two infrastructures, and it is these differences that are described in more detail below.

Policies

The Akenti policy is distributed and hierarchical. It comprises two components: Use-Condition certificates and Policy Certificates. A Use-Condition certificate places requirements on the attribute certificates (ACs) and public key certificates (PKCs) that users must have in order to gain access to a resource. A Policy Certificate states the overall policy for controlling access to a resource, and holds the trusted Certification Authorities (CAs) and Stakeholders, and pointers (URLs[1]) for searching for Use-Condition certificates that are applicable.

Policy Certificates comprise a root policy certificate, and optionally subordinate policy certificates that inherit from the root policy. Akenti sees the target as a tree of resources, e.g. a filesystem with subdirectories. Each of the branches (subdirectories) can have its own sub-policy, but in addition the policy of the superior branch (directory) is inherited. Each of the policies can be issued by different Stakeholders.

A Stakeholder is a special kind of authority that is trusted to issue Use-Condition certificates and subordinate policies. Each stakeholder can impose his own access control requirements independently of other stakeholders, but note that this can lead to one stakeholder blocking access to all users (see later). One of the stakeholders signs the Policy Certificate. A stakeholder in Akenti is equivalent to a Source of Authority (SOA) in PERMIS.

[1] http:, https:, ldap: and file: protocols are possible, thus enabling storage of the certificates in Web directories, LDAP directories and filestores. However, Akenti does not specify the LDAP schema for storing their UCCs and Attribute Certificates, which thus makes LDAP effectively unusable.

Use-Condition certificates contain the name of a target resource, conditions on its use, a critical flag[2], the trusted issuing authorities (CAs for X509 PKCs and Attribute Authorities (AAs) for Akenti ACs), plus a list of rights/privileges that are granted if the conditions are satisfied. Conditions say which attributes a user must have, and can include identity attributes from PKCs (e.g. CN), role or group memberships (e.g. groupX), attributes from ACs, and environmental parameters (e.g. time of day). Rights are comma separated lists of actions on targets. Action names have to be unique for the whole domain of resources, irrespective of the target type. By way of example, a stakeholder can specify that in order to read or execute a process on a target the user has to possess a CN attribute with a value matching one from a given list (thus modelling DAC), signed by a particular CA. A detailed description of a Use-Condition certificates can be found in [UCC] [AkentiCerts].

The attributes issued in ACs, are independent of each other and cannot form a role hierarchy (i.e, in which superior roles inherit the privileges of subordinate roles). This is one difference between Akenti and PERMIS.

An Akenti Policy Certificate comprises: the name of the resource to which this policy applies, information about the trusted CAs (their names, PKCs, and locations of their repositories), the trusted stakeholders and URLs to search for Use-Condition certificates, the caching time[3], and optionally URLs to search for user ACs.

The whole policy is signed by one of the stakeholders and must be stored securely to stop it being switched for another one (or deleted altogether). Similarly, policy hierarchies are not specified in a secure way, since there are no pointers from superior to subordinate policies or vice versa. Consequently the policies must be stored in secure directory hierarchies, and the directory hierarchy determines the policy hierarchy. This is another major difference to PERMIS.

A full description of the Akenti policy can be found in [AkentiPolicy] [AkentiCerts]

The PERMIS policy is one object, and is stored in an untrusted LDAP directory as a policy attribute certificate. It supports classical hierarchical RBAC, in which roles are allocated to users and privileges to roles. Superior roles inherit the privileges of subordinate roles in the hierarchy. Multiple disjoint role hierarchies can be specified. PERMIS has a very loose definition of a role; a role may be any attribute assigned to a user, not just a conventional organisational role. PERMIS supports the distributed management of attribute certificates, and multiple external SOAs can be trusted to issue roles/attributes. Thus users can be certified externally to the domain the attribute

[2] If the UCC is Critical (flag set to true), the UCC must be satisfied, in the sense that if the condition fails to be satisfied, no access can be granted at all, irrespective of what other UCCs allow. If the UCC is Non-critical (flag set to false), the UCC only defines rules for one access control condition and it does not affect the decisions made by other UCCs.

[3] This is the maximum time in seconds that any certificates (PKCs, ACs) used by the Akenti server may be cached.

certificates will be used in. The policies are kept in the LDAP entry of the policy issuer, and different policies are distinguished by their unique object identifiers (OIDs). There is no need for the policies to be kept securely, since the PERMIS engine validates the policy at run time to ensure it is the correct one (i.e. has the correct OID and is signed by the SOA). However, the name of the SOA and policy OID have to be securely configured into the PERMIS application at start up. Policy hierarchies are not supported by PERMIS. These can either be enforced organisationally by management, or by the application instantiating several PERMIS decision engines, one per level of the hierarchy, and ensuring that each level grants permission.

The PERMIS XML Policy comprises:

?? The unique policy OID, so that different policies can be identifiedo
?? The Subject Domains, which are the subjects who can assert roles
?? The Role Hierarchy policy, which lists the roles and their hierarchical relationships
?? The SOA policy, which lists the trusted SOAs
?? The Role Assignment policy, which says which SOAs are trusted to issue which roles to which subject domains, and whether delegation is supported or not
?? The Target Domains, which lists the targets governed by this policy
?? The Action policy, that lists the known actions and their parameters
?? The Target Access Policy, which specifies the set of roles/attributes required to perform a particular action on a particular target, along with any conditions.

A full description of the PERMIS policy can be found in [IFIP].

Policy Conditions

In PERMIS conditions are placed on which attribute certificates can be trusted (in the Role Assignment Policy) and on which attributes have which privileges and when (in the Target Access Policy).

In Akenti conditions are placed on which attribute certificates can be trusted and on which attributes have which privileges and when (in the Use-Condition certificates).

PERMIS therefore contains a level of indirection, in that principals are assigned attributes, and attributes are given privileges (i.e. classical hierarchical RBAC). Akenti however can support DAC and $RBAC_0$, in that principals can be given privileges or group membership, and group attributes can be given privileges.

Distributed Management of Trust

Both PERMIS and Akenti have a Source of Authority (SOA) or equivalent entity that creates a Policy. PERMIS uses the X.509 terminology SOA, Akenti calls it a Stakeholder. In PERMIS, the application gateway is securely configured with its SOA name (authorisation root of trust) and the unique object identifier of the policy to use. The PERMIS ADF reads in the policy, checks it is signed by the SOA and contains the object identifier, and then knows it has the trusted correct policy to work with. The policy contains the names of remote SOAs who are trusted to issue attribute certificates.

In Akenti there can be multiple stakeholders participating in administering the resource. All the stakeholders can issue Use-Condition certificates, and one of them creates and signs the root policy and places it in a secure store. The root policy lists the other stakeholders who are trusted to issue subordinate policies and Use-Condition certificates. Use-condition certificates state which AAs are trusted to issue which attribute certificates.

Both PERMIS and Akenti recognise separate hierarchies for authentication (CAs) and authorisation (SOAs or stakeholders). Both infrastructures must be configured with the CA (authentication) roots of trust. In PERMIS it is an application dependent matter how this is configured into the system. In Akenti it is part of the policy.

Attribute certificates contain the distinguished names (DNs) of their holders (users). In PERMIS, the application must properly authenticate the users and validate the digital signatures on the attribute certificates. A user must authenticate himself to the application to prove that he is identified by a given DN, and then PERMIS can trust that attribute certificates containing this DN belong to the user.

In Akenti, users who present attribute certificates must eventually digitally sign something to prove that they are the holder of the private key corresponding to the public key held in the PKC referred to in the attribute certificate (the AC holder is referred to by his DN and the DN of his CA issuing his PKC). Akenti can then trust that this attribute certificate belongs to this user. The attribute certificate is further checked to ensure that it is signed by a trusted AA, and that it conforms to a trusted Use-Condition certificate.

Attribute Certificates

In both systems ACs are issued to users, and hold their privileges (either directly or indirectly via an attribute/role). Akenti uses XML certificates in their own proprietary format. (Note that this format has changed between releases, so that V1.1 wont work with V1.2). The certificates can be stored in LDAP, HTTP or a file repository (but since no LDAP schema is defined this effectively rules out the use of LDAP). A user is identified via his LDAP DN and the DN of the CA issuing his public key certificate (and he has to prove ownership of the private key corresponding to this public key certificate). Attributes comprise a type and value.

PERMIS uses DER-encoded attribute certificates in X.509 standard format. A user is identified by his globally unique X.500/LDAP distinguished name, and a user has to be authenticated against that name. Attributes have a type and value, and attributes can form arbitrarily complex role hierarchies. The certificates are stored in LDAP repositories, using standard LDAP/X.500 schema. The base code can be extended to support other repositories with LDAP-based naming conventions.

Decision Making

PERMIS operates in multi-step decision making mode. In step 1, *getCreds*, the user’s ACs are obtained and validated, and the roles that conform to the policy are passed back to the calling application for caching. This typically takes place during user login. In step

2, *decision*, the requested action and target are passed, along with the user's validated roles, and a simple Boolean decision is returned, either granting or denying access. Step 2 can be repeated as often and as many times as required for different targets and different actions, as the user attempts to perform different tasks.

Akenti only operates in single step decision making mode, but is able to make different types of decisions. The client can ask "What can a user do?", as well as the traditional "Can this user perform this action on this target?". Akenti always embeds its response in a Capability Certificate. A Capability Certificate says what a user is allowed to do. It comprises the public key of the user, the DN of the user and his CA's DN, the name of the resource, and the privileges that the user enjoys, optionally with a list of conditions attached to each of them. The Capability Certificate is then signed by Akenti and given to the client. The client can be either a gatekeeper or the user himself, and if the user, (s)he can subsequently present this to a gatekeeper. The gatekeeper, which must hold the Akenti public key, merely needs to check the signature on the capability, then ask the user to sign a challenge, before granting (or denying) the user access to the resource.

PERMIS has no ability to return Capability Certificates.

Akenti also supports caching, but this is internal to the Akenti engine. Certificates are read in, validated and then cached for use in subsequent decision making. A parameter in the policy says how long the cache is valid for.

Software Architecture and Sizes

Akenti comprises C++ classes and dynamic link/shared libraries. It has a set of modules, some of them are for the standalone server, some for the API implementation that can be embedded directly into a gatekeeper application. The standalone server can receive authorisation requests via the network over an insecure connection or SSL. The Akenti web-security module can be attached to an Apache web-server for Unix platforms.

PERMIS is an API implementation only, written in Java. The Java classes are to be used by the gatekeeper Java program. There is no standalone authorisation module at the moment (although one is currently being built).

We built two test programs, one for the C++ API, one for the Java API. The PERMIS test program with the decision engine is a JAR file of approximately 200KB, which makes it about 900KB with all the necessary libraries (XML parsing, cryptography etc.). On top of this is the size of the JRE, which is about 130MB for IBM's JVM, but this can be significantly reduced by removing all graphics related modules, if only the PERMIS engine is used on the computer. Note that Sun's JVM is at least twice as small, but we did not run tests on that JVM.

The Akenti executable file is about 16MB (compiled for Linux), plus it requires additional shared libraries, the total size of which amounts to another 6MB. The Akenti developers report that the engine is even bigger on Solaris computers and may exceed

50MB [Private Correspondence with LBL]. Note that Akenti also needs a JRE for certificate signing.

Administration: Allocating Privileges and Setting Policies

In Akenti there is a special command line tool for creating the Policy, Use-Condition and Attribute Certificates. A GUI tool can also be used, and if a Resource Definition Server is running this will ensure that the administrator conforms to the policy when issuing ACs.

In PERMIS, there is a GUI application, the PA (Privilege Allocator) that is used to create and sign policy ACs and user ACs and store them in an LDAP directory. There is also a programmable API that can used for the bulk creation of ACs. The PA and API will issue any type of AC, and it is the administrator’s responsibility to ensure that the contents are correct.

Ease of Installation and Use

Installing and Using PERMIS

Two installation cookbooks are provided with the PERMIS implementation: a Privilege Allocator Cookbook and a PERMIS API cookbook. The authors have received written installation reports from 3 sites in the USA: the University of Alabama, Michigan University and the Akenti developers at Lawrence Berkeley Laboratory. These are available from the Salford PERMIS web site [Permis].

No site found any significant problems in downloading and installing the software. Minor errors or omissions were found in the documentation and these have been corrected. One site reported problems when working with iPlanet Directory Server v5.0. It does not support LDAP requests that use object identifiers instead of attribute type names, in contravention of the LDAP standard [LDAP]. Consequently we have produced a work around for this.

Whilst no sites reported operational use of PERMIS, never the less they did create their own policies and test them with the test application provided with the release. Alabama reported that they would have significant overhead in the management of roles and ACs, since these are very dynamic in their environment. Therefore a set of management tools would be needed before PERMIS could be used operationally.

Installing and Using Akenti

We had significant problems installing and using Akenti. A few examples follow.
No specifications about Linux versions were given. We had problems trying to compile the code on Linux Red Hat 8.0 and with running the JRE provided on Linux Red Hat 8.0, and therefore could not create the certificates. Whilst the Akenti instructions are quite detailed, not all of them reflect what the code actually does. There were no instructions about the Apache configuration and the Akenti engine refused to work with Apache 1.3.24 running several Virtual Hosts and could not retrieve remotely stored certificates

from it. There were no instructions about the naming convention for the certificate and Akenti refused to pick up seemingly correct certificates.

There are a command-line tool and a GUI tool for creating a policy. Writing a basic policy is easy, but there were several problems with using the program itself, which caused a delay of over two weeks in the testing and required the Akenti team to release several updates of their binary distribution. Many problems were experienced with the GUI tool, such as: the JRE provided with the binary distribution failed to run on Linux Red Hat 8.0; it ignored some of the configuration parameters, like the name of the policy file, when creating Use Condition Certificates; it failed to sign the policy and certificates when running the IBM JVM instead of the Sun JVM; and the initial version of the GUI (downloaded after 25 April 2003) generated a wrong policy (generated 'AND' instead of '&&', so the engine failed to understand the policy).

Performance Testing the Akenti and PERMIS APIs

Test Environment

Two kinds of test were performed: Basic and Medium, the level being defined by the complexity of the authorisation policy. In each test a number of access requests were issued and the following resource consumptions were measured:

- time it takes to collect and verify the subject's credentials (Akenti ACs, PERMIS ACs – a call to *getCreds*). We also measured the time it takes PERMIS to perform a call to the *decision* method. The Akenti engine does not have an equivalent two-step decision process so this measurement could not be taken.
- Memory used during the above processes
- CPU usage statistics

The requests were fed into two specialised programs embedding the respective APIs. The requests were formed as a text file containing text input for the programs, i.e. User DN, Target Name, Action Request, etc. The programs processed the input files and output the statistics onto the standard output, which was then parsed into MS Excel. Appropriate statistics were then calculated.

The PERMIS API was tested in two modes: a) no signature verification on the ACs; b) full signature verification on the ACs, using the same PKI as Akenti. This was to see the effect of cryptography on performance. The Akenti API was tested with cryptography switched on and off using the Akenti configuration parameter (see later). Users were authenticated with no cryptography involved (i.e. the usernames were simply provided), because the PERMIS API allows any user authentication and it would be unfair to require Akenti to use cryptography for this purpose. Akenti was also provided with the CA name as well.

The tests were performed on a PII 500MHz, 256MB RAM, 20GB SCASI hard drive PC. The operating System was Linux Red Hat 8.0, kernel 2.4.18, and the JRE was IBM 1.4.0.

PKI and PMI

Both authorisation engines were configured with the same PKI and similar PMIs.

The PKI consisted of the Root CA that directly certified all the entities participating in the authorisation process. These were: the root authorisation authority (Stakeholder in Akenti, SOA in PERMIS), other authorisation Attribute Authorities (Stakeholders and AAs in Akenti, multiple SOAs in PERMIS), and the Akenti engine. The PKI entities were issued with an RSA 1024-bit key pair, using OpenSSL free software. The distinguished name of the Root CA was cn=Root CA, o=PERMIS, c=GB. The Akenti engine was given a key pair, issued to cn=Akenti Server, o=PERMIS,c=GB

Other subjects and authorities were created as required by the policies and their PKCs were kept in the LDAP entries of their holders. In both test cases the PKCs had to be retrieved from a remote machine for a fair comparison, even though the retrieval algorithms may differ.

The Akenti PMI consisted of one Stakeholder for the Basic tests, and of multiple Stakeholders for the Medium tests. This was intended to measure how the number of stakeholders affects performance. There were as many Attribute Authorities as required for the respective policy. The Akenti Policy always contained only one CA, the root CA.

The PERMIS PMI consisted of one root SOA. There were as many additional SOAs as required for the respective policy.

LDAP and Web

The PERMIS ACs and PKCs and the Akenti PKCs were stored in an LDAP server. It was not possible to store the Akenti ACs in an LDAP server as we were unable to determine the schema required for this. Instead, the Akenti certificates were stored on a Web server as separate files in a web directory.

The LDAP and Web servers ran on the same Linux machine - a P166MHz with FPU computer with 128MB RAM, 2GB+10GB IDE hard drives, 10MB Ethernet card. It had Linux Red Hat 7.2 installed on it. The Web server was Apache Web-server v1.3.24. The LDAP server was OpenLDAP version 2.0.23. The Linux machine was in the same segment of the network (under the same router) as the PC running the test programs.

The performance measurements were taken at different times during the day to account for the varying amount of local traffic, which may affect the LDAP and Web servers performance.

The user entities of the tests were entries with the pmiUser objectClass. Their Attribute Certificates were kept in the attributeCertificateAttribute for PERMIS. It was not possible to store Akenti ACs in LDAP. The SOAs, Stakeholders and Attribute Authorities were also entries with pkiUser objectClass and had their PKCs kept in the userCertificate attribute.

Simple Policy

The simple test policy is for controlling access to one server in a department (e.g. a print server). The implemented policy was:
All staff in the department can write files to laser printer x, Jim the administrator can write files, delete any files from the print queue, pause the printing, and resume the printing at the laser printer x. No-one else is allowed access to the printer.

Medium Policy

The medium policy was provided by the Akenti project at LBL. It is the policy for controlling access to the Advanced Light source at LBL. It states:
The director of the lab wants to make sure that citizens of Iraq, Iran and North Korea can't touch the Advanced Light source, no matter what. The director of the facility wants to be sure that everyone who touches it has taken and passed the lab's X-ray safety training course. The PI for the project wants to allow access for his group members between the hours of 8am-8pm PST and for his colleague's group between 8pm and 8am PST. The role of leader is allowed to control, operate and observe experiments, the role of experimenter is allowed to operate and observe experiments and students are allowed to observe experiments.

The Tests

Statistics were collected for various configurations of the PERMIS and Akenti engines. 120 access requests were issued for the Simple Policy and Medium Policy[4].

The PERMIS engine was tested in four modes, thus producing four sets of results:

1. L+C, storing ACs in LDAP and performing cryptographic checks on the ACs
2. L-C, storing ACs in LDAP but performing no cryptographic checks on the ACs; this can be compared to sample 1 to see how much of the time is devoted to cryptography
3. C-L, performing cryptographic checks on the ACs, but the ACs are stored in memory and are loaded at initialisation time; this can be compared to sample 1 to see how LDAP operations affect performance of the system
4. –C-L, performing no cryptographic checks on the ACs, and the ACs are stored in memory.

The times were taken in four modes for *getCreds* only[5]. The time spent on *decision* is not affected by the mode of operation, and the average for it is calculated over all samples. The same sequence of requests were issued to the Akenti engine in six modes (but only the Simple Policy mode was tested). These were:

[4] Due to the problems with installing Akenti we only succeeded to collect Simple Policy test results for Akenti. Further, Akenti does not evaluate any SYSTEM attributes, like time, and it does not have plug-in mechanism yet, and so is unable to give a simple granted/denied response for the Medium Policy.
[5] In fact, raw data output from the performance tester contains time and memory measurements for both *getCreds* and *decision*. However, statistical analysis of the time was performed in the four modes only for *getCreds*, since the values for *decision* were not affected by the mode.

1. L+C, when the identity certificates (X.509 PKCs) are stored in LDAP, but the rest of the certificates are stored on a web-server, and the returned Capability Certificates are signed
2. R+C, when all certificates (including identity certificates) are stored on a web-server, and the returned Capability Certificates are signed
3. C-L, when all certificates are stored on a local hard drive, and the returned Capability Certificates are signed
4. L-C, the same as 1, but the returned Capability Certificates are not signed (setting in the configuration file)
5. R-C, the same as 2, but the returned Capability Certificates are not signed
6. -C-L, the same as 3, but the returned Capability Certificates are not signed

Note that switching the cryptography off in Akenti is not the same as switching cryptography off in PERMIS. In PERMIS it means that no signature verification on any attribute certificates is done. In Akenti it only affects the creation of the capability certificates sent as the reply of the Akenti engine. If cryptography is on, the capability certificate returned by the Akenti engine is signed prior to returning to the client. If cryptography is off, the capability certificate returned by the Akenti engine is unsigned, and the signatures on the capability certificates are not checked. (This setting also affects the signing and signature verification of cached certificates.) It is not possible to completely switch cryptography off in Akenti without editing the C code and recompiling. Hence the signatures on all certificates are always checked.

The sequence of these tests is repeated with Akenti engine caching switched on and off.

The measurements of PERMIS and Akenti that can be meaningfully compared are:

1. **Single Decision Making**. PERMIS L+C to Akenti L-C and R-C with caching off In this case PERMIS retrieves all of its certificates from a remote LDAP site, performs all cryptographic checks, and returns the internal representation of the credentials. Akenti in mode L-C and R-C is doing similar tasks only fetching certificates from an LDAP or Web server. Switching cryptography off in Akenti means that the engine will not sign the returned Capability Certificate, which is approximately the same as a PERMIS return, but will check the signatures on the retrieved certificates.
2. **Multiple Decision Making**. PERMIS *decision* to Akenti -C-L with caching on (minus the time for the first call). In this case PERMIS has validated the credentials (via *getCreds*) and returned validated roles to the application for caching. The application can then call *decision* multiple times for the same user. Akenti has validated the credentials the first time they are used, then caches them internally, and uses the cache for subsequent decision making for the same user.

The behaviour of the engines in the rest of the tests is incomparable. The measurements for these tests are provided to illustrate how the engines behave under various circumstances.

The Test Results

There were significant problems when trying to use Akenti (see later). The problem was aggravated by the 8 hours time zone difference between the testers and the developers.

The testing programs and raw results are available at [Raw]. The tables below simply provide the average and standard deviation values, with no other manipulation of the raw data.

PERMIS Results

In the Simple Policy, the average number of certificates processed per request is 1.6 ACs (48 requests for Jim with 4 ACs, 3 of which are redundant; 66 requests for Adam with no ACs, 6 requests for Sarah with no LDAP operations)

Table 1. PERMIS Simple Policy Test

	L+C	L-C	C-L	-C-L
Initialisation time(seconds)	1.521	0.833	1.047	0.788
Time for getCreds, average (ms)	142.217	73.070	59.834	25.954
Time for getCreds (ms), std deviation	289.842	189.559	157.875	71.319
Memory used, average (system units)	2621.77	1775.17	2511.16	1627.73
Memory used (system units), stdev	2.51	6.80	4.69	5.08
Ratio of System to User CPU time	0.025	0.014	0.020	0.013

Notes: system unit for measuring memory is a Page of memory. A memory page size was 4KB (which can vary on different systems).

Table 2. PERMIS Medium Policy Test

	L+C	L-C	C-L	-C-L
Initialisation time(seconds)	1.807	1.315	1.473	1.491
Time for getCreds, average (ms)	219.124	87.062	118.342	59.529
Time for getCreds (ms), std deviation	378.320	190.200	207.662	96.375
Memory used, average (system units)	2623.63	1777.82	2522.88	1649.47
Memory used (system units), stdev	2.70	5.53	1.82	5.47
Ratio of System to User CPU time	0.021	0.013	0.017	0.012

In the Medium Policy, the average number of certificates processed per request is 3.4 ACs (9 requests for Bob with 2 ACs, 21 requests for Jim with 4 ACs, 1 of which is redundant; 40 requests for Judy with 4 ACs, 48 requests for Sharon with 3 ACs, 2 requests for Sun Yatsen with no ACs and no LDAP operations).

The number of ACs has risen 2.1 times (110%). This correlates with the rise of time for tests with ACs stored in memory (59ms=100% for C-L, 33ms=126% for -C-L). The tests with remotely stored ACs shows a smaller increase because the system is waiting for a reply from LDAP, which remains approximately the same in both the Simple Policy and Medium Policy tests. (77ms=54% for L+C, 14ms=19% for L-C; the latter has increased less than the former, because L+C makes extra requests for PKC retrieval)

The very large standard deviations in the times for getCreds is thought to be due to Java garbage collection, which runs automatically and periodically when the JVM determines that it is necessary to do so. For example, times for Simple Policy L+C ranged from 6ms to 1.9secs, and for -C-L ranged from 0.8ms to 663ms.

The memory consumption figures are approximately the same (compare them by column) for both the Simple and Medium Policy tests because they have been measured for the whole Java process, which has its own memory management routines. The standard deviation of memory usage, however, is quite different in almost all categories. The increase of memory usage deviation in the L+C tests means that in fact these operations tend to allocate and free big chunks of memory, and that in the Medium Policy test these chunks are slightly bigger. The decrease of stdev for the L-C tests means that a lot of the memory in the L+C tests is used by the cryptographic routines, so an increase in the number of ACs does not cause an increase in the number of cryptographic operations and lessens the memory reallocations. The deviations of memory usage in the tests for locally stored ACs shows a significant amount of memory management operations in the Simple Policy test (a lot of memory is allocated temporarily), and that in the Medium Policy test memory is allocated for longer periods of time (JVM reuses the same chunks).

Akenti Results

Table 3. Akenti Simple Policy Test, Caching off

	L+C	R+C	L-C	R-C	C-L	-C-L
Initialisation time(seconds)	0.012	0.013	0.012	0.012	0.073	0.012
Time for checkAccess, average (ms)	775.6	368.2	700.5	368.8	117.6	116.1
Time for checkAccess (ms), std deviation	1024.8	40.1	1407.5	47.5	23.9	11.1
Memory used, average (system units)	894.98	879.98	894.98	879.98	846.98	846.98
Memory used (system units), stdev	0.18	0.18	0.18	0.18	0.18	0.18
Ratio of System to User CPU time	0.046	0.047	0.044	0.046	0.024	0.024

It is interesting to note that when Akenti accesses the LDAP server as well as the Web server, its performance halves. It is also interesting that switching cryptography on (signing the returned capability certificates) seems to have little effect on its performance, (in fact for R+C performance increased slightly) except when accessing LDAP (L+C), which is very unexpected.

Table 4. Akenti Simple Policy Test, Caching on

	L+C	R+C	L-C	R-C	C-L	-C-L
Initialisation time(seconds)	0.012	0.040	0.012	0.012	0.012	0.012
Time for checkAccess, average (ms)	23.6	17.2	11.4	12.1	11.3	11.9
Time for checkAccess (ms), std deviation	85.5	51.0	22.5	25.5	23.0	26.0
Memory used, average (system units)	900.98	885.98	831.25	831.25	831.25	831.25
Memory used (system units), stdev	0.27	0.18	101.72	101.72	101.72	101.72
Ratio of System to User CPU time	0.020	0.024	0.020	0.025	0.020	0.020

There is almost no difference in the times for four of the six modes with caching on. This is because the URLs are not being used and the certificates are held in the cache. The increased time for L+C and R+C must be due to signing the returned capability certificates.

Memory usage figures remain almost the same for all tests, and show little deviation from the mean.

Table 5. Akenti Simple Policy Test, Caching off, Optimised

	C-L (table 3)	C-L (optimised)
Initialisation time(seconds)	0.073	0.080
Time for checkAccess, average (ms)	117.6	61.542
Time for checkAccess (ms), std deviation	23.9	29.252
Memory used, average (system units)	846.98	833.99
Memory used (system units), stdev	0.18	0.09
Ratio of System to User CPU time	0.024	0.039

At the beginning of May we shared our preliminary performance results with the Akenti developers. They were surprised with the figures and revised their binary distribution accordingly. They provided us with a version of the engine recompiled with better optimisation. Table 5 shows how much this optimisation affects the performance. We did not have enough time to perform all the other tests and only did tests for locally stored certificates (both PKCs and Attribute Certificates). The share of User time to overall time to make one *checkAccess* call has obviously decreased (signs of optimisation) as reflected by the Ratio of System to User CPU time. The overall time it takes to make one *checkAccess* call has decreased approximately by a factor of two. However, we do not expect that network-based times will improve quite as significantly, due to the possible network and server delays.

Comparison of the Performance Results

Comparing the PERMIS and Akenti performance figures, for single decisions we should choose the L+C test for PERMIS (Table 1) and the L-C and R-C tests with caching off for Akenti (Tables 3).

PERMIS time is: 142ms
Akenti L-C: 700ms   Akenti R-C: 368ms

PERMIS performed 2.6 times faster than Akenti in comparable categories, which is a surprising result, considering that the PERMIS API runs on a Java Virtual Machine that interprets the code, and the Akenti engine is written in C++ and therefore is compiled directly into CPU instructions. However, PERMIS consumes 2.9 times more memory.

For multiple decision making we choose the average time for the PERMIS *decision* method compared to Akenti -C-L with caching on (minus the time for the calls to retrieve the credentials into the cache) (Tables 3 and 4).

PERMIS *decision*: 2ms
Note. This is the average time taken over all tests. The minimum time was <0.3ms and the maximum time was 168ms[6]
Akenti (cache only): 6.2ms (excluding the calls to refresh the cache)

PERMIS *decision* with 8 *getCreds*: 7.8ms
Akenti -C-L: 11.9ms (including the time for the calls to retrieve the credentials into the cache)

Note that PERMIS supports multi-step decision making, whilst Akenti does not. With multi-step decision making, PERMIS assumes that the AEF will cache the results returned by *getCreds,* and that the AEF will return these to PERMIS for it to deliver several subsequent decisions. This behaviour is comparable to Akenti with caching on (but in PERMIS the AEF does the caching, whilst in Akenti, it does the caching).

In order to make the figures truly comparable it is necessary to either add the initial time for retrieving the credentials in PERMIS (call to *getCreds*) or to subtract the times of the calls in Akenti to fetch the credentials into the cache. During the 120 tests, Akenti appeared to make 8 network calls to refresh the cache, and when these are subtracted from Table 4, the average time for Akenti cached decision making is 6.2ms. Alternatively, the average time of a call to *getCreds* in PERMIS (C-L) (Tables 1 and 2) is 60ms or 118ms. If 8 of these calls are added to 112 *decisions*, the PERMIS average becomes 7.8ms. Thus PERMIS still outperforms Akenti when multiple decisions have to be made.

[6] The large standard deviation in the times is thought to be due to garbage collection by the Java VM.

Akenti Medium Policy Considerations

When we started implementing the Medium Policy we found that it was very difficult to build an appropriate Policy and UCCs to implement the desired behaviour. We found that in many cases it required a UCC to be built that did not contain any actions associated with it, but only a condition that must be satisfied, for example, a UCC issued by the director that says that if the country is not Iran or Iraq the user is acceptable (but is not actually granted any access). However, the Akenti policy language appears to be deficient in this capacity and does not provide such a capability. The Akenti developers suggested a work around in which the UCC allows a "default" action (that does not actually do anything), and is marked critical. This will then deny all access to any user from Iran or Iraq. Another UCC for Jones' group, with role of student and time between 8am and 8pm is allowed an action of "observe". Now someone who satisfies both UCCs will be allowed actions of "default" and "observe". The application then has to be configured to ignore "default" actions and make use of the "observe" action. Needless to say, we did not think that this was an appropriate or intuitive solution.

The developers claimed in private correspondence that their command-line tools do support the creation of UCCs without actions, and that their policy engine supports this, but we did not have time to check this and have based our conclusions on the documentation and capabilities of the GUI tools that were provided. The developers have subsequently stated that the GUI will be updated to allow for the creation of UCCs without actions.

When we investigated alternative ways of specifying the required conditions, we found a major design deficiency that does not in fact allow distributed management of the resource as claimed in the Overview of Akenti [AkentiOverview]. The deficiency is based in how the UCCs are combined to derive a decision function, and in fact requires that the Stakeholders must communicate their needs to each other in order to create UCCs with joint conditions. We therefore lose the essence of Akenti's claimed distributed management of resources – there is no point in having many issuers of the conditions, if they have to co-ordinate their wishes with a central or superior Stakeholder in order to produce their UCCs, and cannot act independently. When we put this point to the Akenti developers they replied that they have a different interpretation of distributed management of resources. They actually mean co-operating stakeholders who can each issue UCCs, but the underlying assumption is that they must co-ordinate on who is going to set policy about which attributes and who is allowed veto power over what. If multiple stakeholders start issuing policy without considering what the other stakeholders are doing, they can easily deny access to everybody. Therefore the stakeholders have to have a minimum amount of collaboration. They are only independent in the sense that they can all create some aspects of policy.

Another serious lack in the system right now is a function that will display all the existing policies and UCCs for a resource, so that a stakeholder can determine the totality of the authorisation policy for a resource. The Akenti developers recognise this deficiency and plan to implement this function as a matter of urgency.

The limitations of the Akenti policy language that do not allow a stakeholder to create a no-action UCC but only with a condition that must be satisfied, and the discussed deficiency of the Akenti concept did not let us build the correct environment for the Medium Policy. Instead we attempted to build a single Use Condition Certificate that encapsulated all the conditions, issued by the Director, but even though theoretically this should have worked, there was a software problem with the UCC generator and we failed to generate such a complex UCC. This bug has been reported to the Akenti developers. In conclusion, after more than one month of trying, we failed to run any Akenti tests with the medium policy.

Conclusions

We have shown both the similarities and differences between two authorisation infrastructures, Akenti and PERMIS. Both have their strengths and weaknesses. However, perhaps due to the relative immaturity of Akenti, we were unable to run a complete set of performance tests on it. Deficiencies in its documentation and design meant that we were unable to build the medium policy testing environment for Akenti. The developers are fully cognisant of our findings, and plan to issue fixes for these in the near future.

Acknowledgements

This project was funded by the UK JISC. A full report of the project is available from [Report].

References

[Akenti] see http://www-itg.lbl.gov/security/Akenti/
[AkentiCerts] see http://www-itg.lbl.gov/security/Akenti/docs/AkentiCertificates1.1.html
[AkentiOverview] see http://www-itg.lbl.gov/Akenti/docs/overview.html
[AkentiPolicy] http://www-itg.lbl.gov/Akenti/docs/PolicyCert.html
[Blaze] Blaze, M., Feigenbaum, J., Ioannidis, J. “The KeyNote Trust-Management System Version 2”, RFC 2704, Sep 1999
[Chadwick] D.W.Chadwick, A. Otenko. “The PERMIS X.509 Role Based Privilege Management Infrastructure”, Proc 7th ACM Symposium On Access Control Models And Technologies (SACMAT 2002), Monterey, USA, June 2002. pp135-140.
[ISO] ITU-T Rec X.812 (1995) | ISO/IEC 10181-3:1996 “Security Frameworks for open systems: Access control framework
[IFIP] D.W.Chadwick, A. Otenko. “RBAC Policies in XML for X.509 Based Privilege Management” in Security in the Information Society: Visions and Perspectives: IFIP TC11 17th Int. Conf. On Information Security (SEC2002), May 7-9, 2002, Cairo, Egypt. Ed. by M. A. Ghonaimy, M. T. El-Hadidi, H.K.Aslan, Kluwer Academic Publishers, pp 39-53.
[Johnston] Johnston, W., Mudumbai, S., Thompson, M. “Authorization and Attribute Certificates for Widely Distributed Access Control,” IEEE 7th Int Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE), Stanford, CA. June, 1998. Page(s): 340 -345 (see also http://www-itg.lbl.gov/security/Akenti/)

[LDAP] Wahl, M., Howes, T., Kille, S. “Lightweight Directory Access Protocol (v3)”, RFC 2251, Dec. 1997
[Permis] see http://www.permis.org and http://sec.isi.salford.ac.uk/permis
[Raw] see http://sec.isi.salford.ac.uk/akenti-permis
[Report] see http://sec.isi.salford.ac.uk/Papers.htm
[Thompson] Mary R. Thompson, S. Mudumbai, A. Essiari, W. Chin. “Authorization Policy in a PKI Environment”, Proceedings of the First Annual PKI Workshop, Dartmouth College, April 2002, pages 137-149. see www.cs.dartmouth.edu/~pki02
[UCC] http://www-itg.lbl.gov/Akenti/docs/UseCondition.html
[X509] ISO/ITU-T Rec. X.509(2000) The Directory: Authentication Framework