SicAri – A security architecture and its tools for ubiquitous Internet usage

---

Deliverable PF3

Specification of the SicAri Architecture

Version 2.0, 28th October 2004

Jan Oetting, usd.de
Jan Peters, Fraunhofer-IGD
Ulrich Pinsdorf, Fraunhofer-IGD
Taufiq Rochaeli, TUD-SEC
Ruben Wolf, Fraunhofer-SIT

Document Information

Corresponding document version on CVS:
`$Id: pf3-specification-architecture.tex,v 1.50 2004/10/27 17:58:16 jpeters Exp $`

Author of this version:
`$Author: jpeters $`

Date of this version:
`$Date: 2004/10/27 17:58:16 $`

Document history:

```
$Log: pf3-specification-architecture.tex,v $
Revision 1.50  2004/10/27 17:58:16  jpeters
Section added for the SicAri context manager (description of the class diagram)

Revision 1.49  2004/10/25 10:23:40  pinsdorf
Review discussion for version 2.0.

Revision 1.48  2004/10/25 06:53:41  rwolf
New title in Section 6.3, Converted Poseidon pictures in Section 4.4.
Added dummy picture in Section 5.5, please replace.

Revision 1.47  2004/10/18 12:11:01  joetting
corrections

Revision 1.46  2004/10/05 16:53:38  rochaeli
- Authentication Manager
- Identity Manager
- Key Manager

Revision 1.45  2004/09/30 07:59:01  rochaeli
*** empty log message ***

Revision 1.44  2004/09/29 15:58:24  rochaeli

- Platform Architecture
- Security Assumption

Revision 1.43  2004/09/28 11:32:51  rwolf
Added sections 'Platform Overview' and 'Outlook'.

Revision 1.42  2004/09/24 11:22:32  rwolf
Changes in section 2.

Revision 1.41  2004/09/23 11:19:58  rwolf
Added schedule and authentication manager discussion

Revision 1.40  2004/09/23 09:22:08  rwolf
Results of the MW/PF discussion.

Revision 1.39  2004/08/31 14:01:08  pinsdorf
Shortened log list ;-)

*** Older entries have been removed from log. ***
```

Contents

List of Figures

1 Platform Overview

Professional usage of todays communication and collaboration infrastructures requires the consideration of appropriate security measures. However, the security features provided by todays services often annoy the users. The overall goal of the SicAri project is the conception and realisation of a security platform and its tools for ubiquitous Internet usage. The SicAri platform supplies a bunch of applications that provide various security services to the user in a transparent, seamless and integrated way.

The purpose of this document is to give technical details about the SicAri platform from a conceptional, as well as, an architectural and implementation view. It introduces the various components of the SicAri layered architecture, describes its tasks, and gives details about the communication and interaction of platform and external components.

The SicAri's architecture has been developed considering various requirements, that have been derived from the SicAri partner's usage scenarios. The requirements cover different aspects, such as design constraints, security requirements, functional requirements, external interface requirements and quality attributes (see [11, 13]). Taking all these requirements into accounts, the SicAriplatform's architecture has been developed.

The following picture gives a high-level overview of the SicAri's generic security architecture (see Figure 1). On the top of the layered architecture, there is the application layer. The applications defined on this layer can be directly used by users of the SicAri platform. The applications themselves make use of the basic and application services of the platform's service layer. On the one hand, these services integrate external databases, legacy systems, and applications; on the other hand, they provide basic security services, such as authentication or access control. The middleware layer is responsible for the secure and seamless integration and communication of applications and services. The SicAri architecture will be detailed in the subsequent sections.

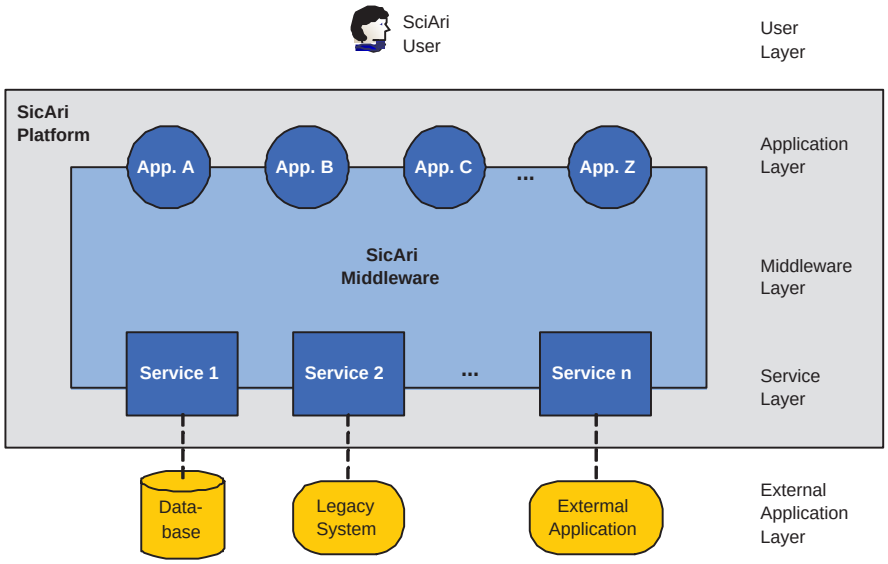

Figure 1: High-level architecture of the SicAri Platform

The remainder of this document is structured as follows. Section 2 outlines the scope of the SicAri platform, while the platform's underlying security assumptions are detailed in Section 3.

Section 4 covers various aspects of the platform architecture, such as an overview of the SicAri terminology, the description of the SicAri kernel and the security architecture, as well as, an outline of the platform communication features. Basic and application services of the platform are introduced in Sections 5 and 6. Sections 7 and 8 provide the logical and the deployment view of the SicAri architecture. Finally, Section 9 provides guidelines and exampled for platform development.

2 Scope of the SicAri Platform

This section describes the scope of the SicAri platform which covers the scope of platform functionality and platform architecture. At the end of this section, the limitations of this platform are also described.

The main function of the platform is to provide interfaces to the user's application to access the services provided by the platform, which are basic services and application services. Together with the middleware, the platform also provides the communication infrastructure between distributed SicAri components.

In case of new users requiring new application services, the platform architecture supports modular and extensible building blocks, adding the possibility to incorporate new services. Therefore, reusability of existing building blocks should also be easy.

The platform should also be able to integrate seamlessly with the existing infrastructure without replacing the established infrastructure. This helps the stakeholders to avoid the unnecessary and cost-intensive upgrade to the information infrastructure.

As the number of user's applications which use the platform grows, the platform is capable to solve the scalability issue by providing two or more platform instances working together within a SicAri infrastructure. This is realised by inter-communication mechanisms provided by the middleware.

All security related aspects of a SicAri platform are regulated by a security policy. In the case of multiple platforms interacting with each other, all platforms running in the same SicAri infrastructure share the same security policy. Each platform has its own policy enforcement component. Access attempts to local or remote services are checked against the security policy considering the requestor's current session ID, activated roles, the available permissions and other parameters.

The SicAri's protocol engineering sub-project developed additional scenarios which consider two SicAri infrastructures interoperating, i.e. there are two SicAri platforms from two different SicAri infrastructures with two different security policies communicating. In those cases, policy negotiation between both infrastructures are required. The purpose of that sub-project is the development of such kind of negotiation protocols. These protocols will be integrated in the SicAri toolbox. The same holds for the scenarios considered by the SicAri's application sub-project concerning ubiquitous computing. Here, the general conditions may also differ from the conditions made in this document.

The SicAri platform and its components use the Java Technology which attracts a broad spectrum of potential users. The Java Technology supply development toolkit ranging from the mobile devices (J2ME), personal computer (J2SE) into scalable, distributed environment (J2EE). The Java development toolkits are also available in many well-established operating systems. To

help the user's application and service developers, the platform interfaces are kept simple and clearly documented with support from various documentation tool such as JavaDoc. The SicAri code conventions and a programming guide are provided as separated documents (refer 9).

In spite of the platform features that are mentioned above, the platform is not intended to replace the existing information infrastructure nor its existing security mechanism, such as firewall, etc.

In the current implementation of SicAri platform, only the J2SE Java platform will be considered. The current limitation of J2ME Java platform to perform cryptographic functions, which carry the important task in the security, impedes the implementation of SicAri platform on the J2ME Java platform.

3 Underlying Security Assumptions

To achieve the desired security goals, the platform has to implement a set of security functionalities. The implemented functionalities are described here briefly. A complete list of required functionalities can be found in the platform requirements document [11].

- **Audit log generation**
  In case of a relevant security event occuring, the platform SHALL log the event in the audit database, along with the information that relates to the event, such as user identity, time stamp, etc.

- **Restricted access to the audit log**
  An audit log is sensitive data that is only available to the authorized user, such as the administrator of the platform. Therefore, the access to the audit log MUST be restricted.

- **Selectable audit log review**
  When the security auditor wants to perform auditing on the platform, he/she MAY be provided with a set of simple data base query functionalities to filter out the unwanted information. This should help the security auditor in analyzing the audit log.

- **Provide basic cryptographic services**
  Cryptographic functions play an important role in security. Such function are: data encryption, data decryption, creation and verification of digital signatures, etc. supporting one or some of the security goals. These functions MUST be provided by the application services, and thereby will be available to the user's application which uses the platform.

- **Role-based access control (RBAC)**
  In current daylife, users tend to work togther in a clear and structured organisation. Each user has his/her own responsibilities to perform the tasks. RBAC fits perfectly into this scene: this mechanism allows the discretion of rights to perform actions (read, write, execute) based on his/her role. With this argument, the RBAC mechanism MUST be integrated into the platform. This mechanism shows some advantages, mostly with respect to efficient administration and policy neutrality.

- **Information flow control/filtering based on security attributes**
  Another important view of information security is the information flow control/filtering. It may happen in some scenarios that access control mechanism alone is not enough to

achieve one of the security goals. While many access control mechanisms put its control mainly on a subject that accesses an object, the information flow control mechanism controls the flow of information based on security attributes carried within the information itself, such as security clearence level embedded into the document as watermark. Therefore, the Platform SHALL provide the information flow control.

- **Failure login detection**
  Suspicious attempts to break the authentication process should be treated in a reasonable way. The authentication mechanism MUST deactivate an user's account in case of too many login failures. This mechanism will minimize the probability to gain unauthorized access by the malicious user.

- **Support password and public-key based authentication**
  The most widely used mechanisms which are used to perform authentication are password-based and public-key-based authentication. Therefore, the platform MUST support these authentication mechanism.

- **Management of user roles and administrative roles**
  In a large scale distributed system, it is a complex task to administrate the roles. Therefore, the platform SHALL provide services to perform user and administrative roles management.

- **Self test of each services**
  To assure the correct functionality of the services, each service MAY have a self test functionality that can be performed periodically or upon request.

- **Management of security function behaviours and parameters of security functions**
  The security requirements of each SicAri's scenario differ each other. With this situation, the platform behaviours should be configurable, adapting to the needed security requirements. Through the policies, the behaviour of platform and its services MUST be managed.

- **Revocation of security tokens**
  Some security tokens, such as certificates, may expire after a specified lifetime, or lose their validity caused by an unauthorized access. To countermeasure this, the authorized administrator SHALL be able to revocate the security attributes.

- **Trusted platform administrator**
  The platform administrator MUST be trustworthy. The administrator MUST have an unique ID and a valid certificate. Therefore, each platform MUST be managed by a trusted administrator, who starts the platform. By having trusted administrator for each platform, the possibility of identity impersonification should be minimal.

- **Prevention of execution of malicious service on the platform**
  The Platform MUST provide mechanism to prevent the execution of malicious service on the platform. To realize this, the User-ID of the administrator is embedded with the Platform-ID and creates an unique User-ID/Platform-ID across the SicAri infrastructure. Together with the code-signing mechanism, the unique User-ID/Platform-ID prevents the execution of malicious service in the infrastructure, specially on the platform.

Nevertheless, we also identified problems that are not meant to be solved by the platform alone:

- **Delegation of rights**
  In a situation where an employee leave for vacation, he needs to transfer his right to another employee that requires a delegation mechanism. The platform currently does not support any delegation of rights.

- **Security in an operating system**
  The built-in security functionality in an operating system should not be replaced by this platform. The platform runs on top of the operating system, providing the user with a secure and higher abstraction level to access the internet ubiquitously.

- **Security in a database system**
  The platform focuses on providing the user with the secure ubiquitous internet access, which means the internal security is leaved untouched by the platform. The database query always take place internally in host. Therefore, the platform can not replace the existing security mechanism in the database.

4 Platform Architecture

The design of the platform's architecture was deduced from both the platform's requirements [11] and requirements on tools to specify, management and enforcement of security policies [13].

Figure 2 shows the layered architecture of the platform. The platform is based on SicAri kernel which runs on the top of the Java Virtual Machine. The kernel has its own **shell**, `environment` and **security manager**, which are explained in 4.1

Additionally, the basic and application services are also integrated into the platform, and provides the user application with the security related services. The user application that runs on the platform could access the services provided by the platform.

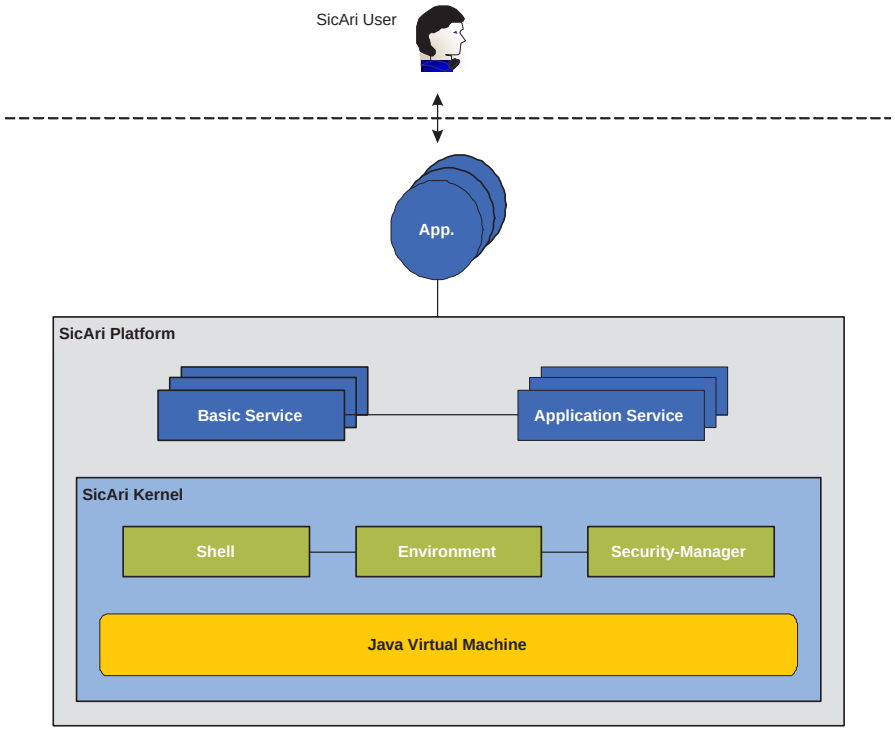

Figure 2: Layers of the SicAri Platform

The SicAri platform offers the following advantages:

- Minimalistic design
  The design of SicAri platform is kept simple. This could simplify the management of the platform and minimize the security issues caused by the complexity of a system.

- Modularity and reusability of services
  The modularity of the platform offers a simplified management of the platform. This concept is applied as the basic and application services in the platform, which can be replaced easily. It is also intended that the services could be used as a standalone module or as an integrated component of any system other than SicAri.

- Extensibility
  Supported by modularity of the services, the functionality of the platform should be easy to extend. The application services, which are optional and depend on each use scenarios, enables the extensibility of the functionality of the platform.

- Maintainability
  The modularized services concept also support for simple maintainability.

- Intrinsic Security features of SicAri kernel
  By considering the security features from the beginning of platform's design process, the platform supports a seamless integration of security functionality.

4.1 Terminology

This sections defines the basic terminology of the SicAri architecture. Whenever one of these terms is used – execept explicitly declared otherwise – it refers to the definition given in this section.

Environment. The environment is a hierachical namespace for object instances. Object instances can be registered, looked up, searched for, and removed. Any of these operations use slash-separated ('/') identifiers, called *path names*, to identify certain object instances.

Shell. The Shell is a user interface for the environment. It allows administration of and access to the environment. Hence, it supports at least the afore mentioned operations *register*, *lookup*, *search*, and *remove* of object instances. Furthermore it allows navigation in the namespace and invoking of Java methods. The SicAri shell compares to a UNIX Shell, where the enviroment stands for the file systems.

Security Manager. The security manager is responsible for policy enforement. The Java programming language provides a standard security manager which is accessible via API. SicAri replaces this security manager with an own implementation.

Service. A service is a software which fullfills a very specific task. It provides a small, well-defined programming interface. Typically there is no direct interaction between the user and a service. The term *module* is used as synonym for *service*. Every service is published as an object instance in the enviroment which allows access to other services and applications. Services are retrieved by means of the environment's search and lookup functionality. The service may be separated in a local accessor stub and one or more remote components which provide the functionality. The service is the way to integrate legacy applications and exernal data sources into the SicAri architecture.

SicAri kernel. The SicAri kernel (or just *kernel*) consists of the Java implementations of the environment, the shell, and the SicAri security manager as defined above. Together they provide service bootstrapping and configuration, local service management (registration/searching), and a consistent security environment by means of implicit access control. The term *middleware* is a synonym for *kernel*.

SicAri platform. The SicAri pattform consists of the SicAri kernel started on top of a Java Virtual Machine a number of mandatory basic services and optional application services. These services are described in Section 5. Any application can rely upon the availability of the basic services. Particularly in the deployment view *platform instance* is a synonym for *platform*.

SicAri infrastructure. The SicAri infrastructure is a compound of several platforms. These platforms may be distributed within the infrastructure.

SicAri application. An application is a software which fulfills a complex task. Since it usually interacts with the user, it provides both an interfaces for user-interaction and a programming interface. Applications make use of services in order to fulfill their tasks.

4.2 SicAri Kernel

As defined in the last section, the SicAri platform is rather a collection of services flexibly loaded and configured within a common environment on tob of the Java Virtual Machine, than a monolitic system designed as one static piece of software. This common service environment is constituted by the SicAri kernel which is charactarized by the following additional features:

- Bootstrapping of basic SicAri services.
- Platform configuration and service dependency checking.
- Service management within a common service directory.
- Uniform security context for local service interaction.
- Secure class loading and thread separation.
- Semantic separation of caller and callee.
- Administration through a shell-based interaction layer.
- Monitoring and loging during runtime.

Illustrated in Figure 3, the main functionlities of the SicAri kernel are *identity management*, *policy enformcement*, and *service module management*.

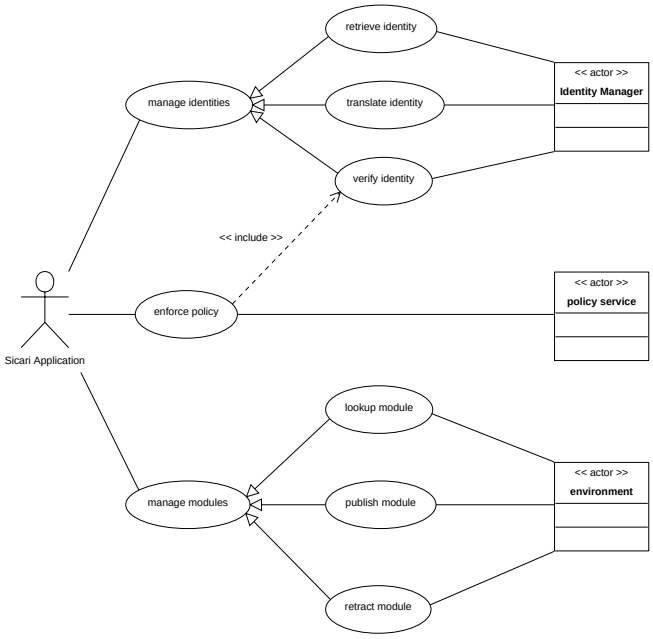

Figure 3: Use Case: SicAri Kernel

Use Cases "SicAri Kernel"
Identity Management:

1. Every user who wants to log into the SicAri infrastructure has to be registered by the identity manager with a user ID.

2. When a user has succesfully authenticated against the local SicAri platform instance, a new session ID is created.
3. Every process within SicAri is associated to a session with a valid and secure session ID.
4. The session ID can be mapped to the user ID via the current session context.
5. Every SicAri platform instance is started by a local administrator, as a special kind of user.
6. Only platforms started on behalf of a registered administrators are accepted as service providers within the SicAri infrastructure.

Policy Enforcement:

1. Access to critical resources is checked using the current user ID as authencication token.
2. Security policies define Role-Based Access Control and allow additional static and dynamic constraints.
3. Policy decision and enforcement is made by the policy service as basic SicAri platform service.

Module Management:

1. The soley kind of service interaction is the use of the environent as local manager for service publishing, lookup and retraction.
2. The use of SicAri functionality is restricted to the use of SicAri services.
3. As well as an applications the kernel does use higher SicAri functionality through loaded platform.

The SicAri kernel consists of three components, namely the *Shell*, the *Environment*, and the Security Context. These are described more detailed in the following.

4.2.1 Shell

The *shell* is an interaction layer for the platform administrator. Comparable to a regular UNIX shell, the SicAri shell provides variables, syntax for basic control sequences, i.e. loops and conditions, and supports a number of built-in commands (see Section 9.3). Thereby, the administrator may load and configure services as well as monitor them during runtime. The same way, the administrator can interact with the SicAri platform through the shell, every shell command can be executed automatically by means of a shell script. Thus, shell scrips allow automatic bootstrapping of the platform.

A dedicated service module implementing a SSH-like server provides remote login to the shell interaction layer through a corresponding client. This client implicitly authenticates the platform instance's administrator with a private key. As illustrated on Figure 4 the client can either connect to the platfrom from the local or a remote hosts (compare (1) and (3)), whereas remote administration is especially intersting, when running the SicAri platform instance as daemon in non-interactive mode.

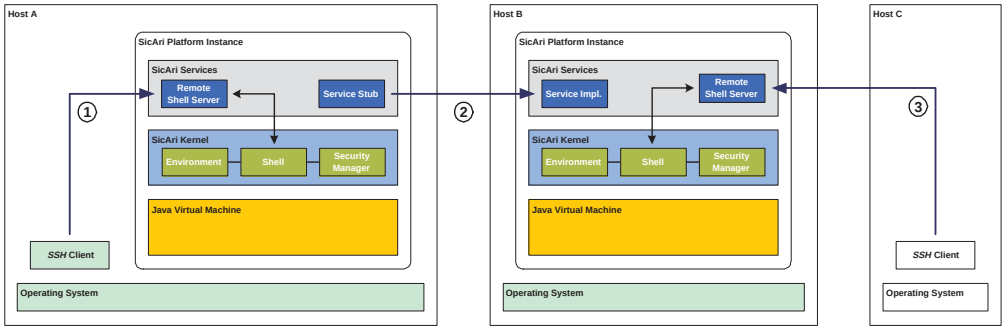

Figure 4: Interactin with and between SicAri Platform Instances

In interactive mode the shell shares the same input, output and error streams with the services loaded within. That's why a so-called `ConsoleFilter` controls these IO streams and prevents services from manipulating or closing the streams intentional or accidentially.

Further, the shell includes explicit commands to load and register services and navigate within the service environment, which is described in the next section.

4.2.2 Environment

Service management is handled by the *environment* component. The environment is a hierarchically structured namespace wherein service instances can be published under a given path. This is analog to a file system, where paths lead to files - with the exception that paths are virtual in the SicAri environment, i.e. the paths of all published services build the hierarchy and not vice versa. Subsequently, this anomaly allows the coexistence of a path and an equal service name within the same "directory".

Again analog to a file system, the access to sub hierarchies or service objects can be restricted by means of access rights, as there is a specific permission for read access (*lookup*), and two permissions for write access (*publish*, *retract*).

Since the environment is the only way of interaction between service and service, kernel and service, or application and service, this is an essential location for policy enforcement.

Besides being a kind of service directory, another important feature of the environment is the encapsulation resp. separation of services. When publishing a service the administrator can specify a so-called *dynamic proxy*, which is used for service interaction, subsequently. The following two types of proxies are specified for the SicAri platform:

Plain proxy This type forwards method invokations to the encapsulated service object within the same thread. Initiated by a privileged signal, the proxy truncates the reference to the encapsulated object and releases it for garbage collection, as there are no more strong references to the object. Further method invokation leads to a corresponding exception thrown by the proxy. Thereby, access to the service can be prevented, even if another service has requested access to the object and still keeps the according reference, since the reference refers to the proxy not the encapsulated service object. Nevertheless, the proxy is not able to appropriately treat references returned by methods of the encapsulated object. Otherwise mechanisms were needed analog to active firewalling.

Asynchronous proxy This type initiates a distinct thread to serve method invokations. The caller thead is blocked till the result of the invoked method is available. The advantage is, that the caller thread may set a timeout, after whose expiry the caller thread can terminate, even if the method result is not available so far.

In both cases only public methods of all implemented interfaces of the published service object are wrapped by the proxy. These mechanisms reduce DoS attacks by possibly malicious or faulty applications and services. Further the encapsulation allows clean security context switches between caller and callee.

Another security feature enables the return of a local view of the global environment to each single service. This local view of the environment automatically remembers all objects published by this service, and then retracts all those object, when the service is terminated.

4.2.3 Security Context

Besides the security mechanisms provided by the shell and the environment the SicAri kernel implicitly and explicitly provides an additional security context for services launched from the shell resp. for users logged in the platform via an application. This security context is the precondition for implicit policy enforcement via the SicAri security manager (see Section 5.1).

On the one hand, services can automatically be launched within their own security context, having their own `ClassLoader` and running within a new `ThreadGroup`, when needed. Generally, services are launched by or on behalf of the platform administrator inheriting his Java security context by means of a set of permissions granted. With this *service security context* it's possible to fundamentally restrict the service's permissions in a dynamic manner.

On the other hand, the *SicAri security manager* implicitly enforces access policies dependent on the user currently invoking platform services (see Figure 5) by delegating the policy decision to a special module. This module needs to know which user is currently accessing the critical resource to request his context and finally grant or deny the corresponding access. As precondition for this process the kernel has to investigate the according user ID.

To solve this problem without explicitly tagging each method invoked by a user with its user ID, a mechanism compared to the service security context can be used: When the user logs into the SicAri platform, its user ID is the result of a successfull authentication. Further, a temporarily session ID is automatically created. As long as the user's actions are executed within its own `UserThreadGroup`, this and all sub threads can implicitly be annotated with the user's session ID by means of a Java `InheritableThreadLocal` variable. This, in turn, enables the SicAri security manager to implicitly use this variable, as long as the security manager is invoked within this user's thread group.

Since Java's AWT and Swing framework is a "bloody mess" according to security and thread separation, it's recommended to clearly separate graphical user interfaces from the rest of the application. AWT/Swing creates one thread group for the event queues upon first invokation, for example. That means, an actions triggered by a click on an window is always executed in the same thread group (of the AWT/Swing event queue) instead of the user's thread. Through clear separation of GUI and functionality, a help class might transfer the security context back into the thread group of the user, which has actually clicked the button.

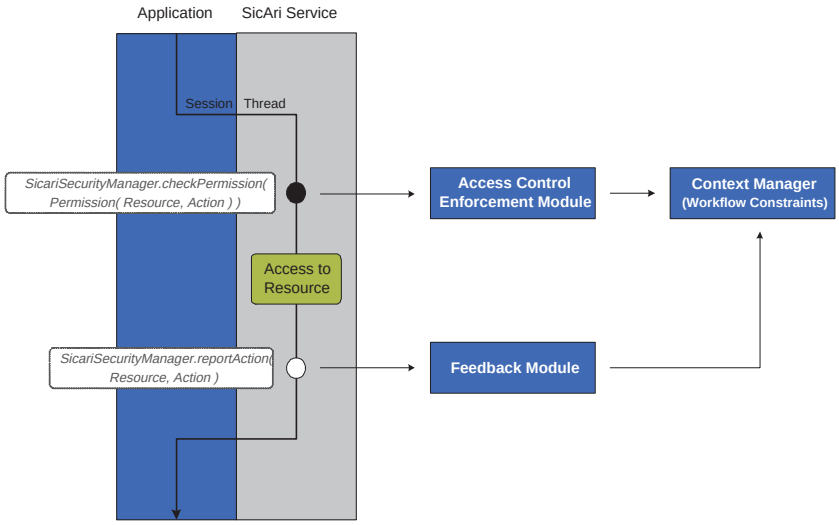

Figure 5: Access Contol Enforcement and Session Context

4.3 Security Architecture of the SicAri Platform

The SicAri security architecture constitutes a collection of software and hardware components that protect and control the use of sensitive resources. These resources may be files or databases, services like e-mail or printing as well as security related information like passwords and certificates. Like most security architectures [8] the SicAri architecture is described by means of application programming interfaces (APIs) and hide the internal details of components. Besides this kind of abstraction, security architectures often use a layered approach, where the lower layers (the SicAri kernel) provide base security features (like key and certificate management, access control, and security policies) to be implicitly or explicitly used by higher level applications and services.

Since security-related functions that handle sensitive data influence the architecture, security must be considered in every aspect of the design. This section gives in overview of the security framework of the SicAri kernel and its subsystems.

4.3.1 Overview

The main security objectives of the SicAri kernel, i.e., identity management, policy management, and service management, have already been introduced in Section 4.2. In this section we give some more details with a special focus on policy management and authorisation.

The SicAri platform specifies a policy for all security related concerns. This policy includes statements with respect to, e.g., confidentiality, such as permissions to access some resource or the necessity to encrypt a certain communication channel. All sensitive operations inside the platform are checked whether they are conform with the underlying security policy. Additionally, components of the platform may ask for aspects of the security policy in order to provide the right service. Requirements for SicAri policies have been analyzed in [13] comprehensively. The specification of SicAri policies in human-readable way can be found in [12]. This document shows, which kind of security policies can be handled within SicAri together with the respective approach.

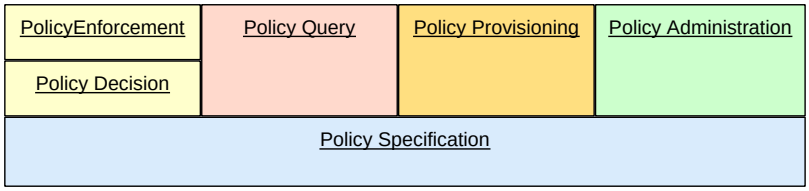

Figure 6: Components of the SicAri Security Framework

The SicAri policy framework is divided into several components: Policy Enforcement, Policy Decision, Policy Query, Policy Provisioning, Policy Administration, and Policy Specification as illustrated in Figure 6. This approach was chosen due to the requirement for the separation of policy mechanisms and the policy specification.The details about the various policy components are given in the subsequent sections.

4.3.2 Authentication and Access Control

The SicAri security framework mainly relies on the two basic mechanisms: authentication and access control. While authentication mechanisms ensure that system users are who they claim to be, these mechanisms say nothing about what operations users should or should not perform within the system. To afford protection to that effect, it is necessary to use access control[5]. Access control deals with determining the allowed activities of legitimate users, mediating every attempt by a user to access a resource in the system. Therefore, the objective of an access control system is often described in terms of protecting system resources against inappropriate or undesired user access.

In any case, the access control component takes a proved identity as basis for its access control decision. Therefore, user authentication needs to take place before access control decision. Section 5.5 outlines the features of the authentication manager.

There are many models and mechanisms for access control. Most of them make use of the terms subject and object. In a computer system, a *subject* is an entity that can initiate a request to perform an operation. In our case, users of the SicAri platform may also be represented, for example, by a process, a threat, a component or an agent acting on behalf of a user. That is, in order to check the permissions of, for example, an agent, the system needsd to find out, on which behalf the agent is acting. An *object* is a system entity on which an operation can be performed. An object is an abstract concept that is useful for purposes of generically modeling access control approaches and describing access control mechanisms. An object may represent an application, a file, a database entry, or system resources like a printer or a network device. Additionally, depending on the access control mechanism used, the system may differentiate the *kind of access* (or *operation*), such as read, write, execute in the case of files. Many mechanisms use the term *permission* in order to describe the pair of object and operation. For example, a system may define a permission to read file X or to print on printer Y.

Various access control mechanisms (such as access control matrices and capability lists) have been developed, additionally there are many access control models for different areas of application and realisation of mandatory or discretionary access control (such as Bell-LaPadula, Clark-Wilson, or Chinese Wall).

Within SicAri, we will not focus on a special kind of security policy. Furthermore, we do not propose a new access control model. We follow the *Role-based Access Control (RBAC)* mechanism. This mechanism has been approved as standard ANSI INCITS 359-2004 in February 2004. Within SicAri, we will extend the RBAC mechanism with respect to consideration of dynamic context information within the policy specification and the access control decision (see Section 4.3.4).

4.3.3 Role-Based Access Control (RBAC)

Since the general concepts of RBAC are well-understood and extensively described in the literature [6, 14, 5], this section only presents the main concepts of RBAC.

The RBAC standard specifies four different functional components with increasing set of features and requirements (see Figure 7). There are several options within each functional component which result in different requirements packages. We will only briefly describe the key aspects. For further information about RBAC please refer to [10, 5, 6].

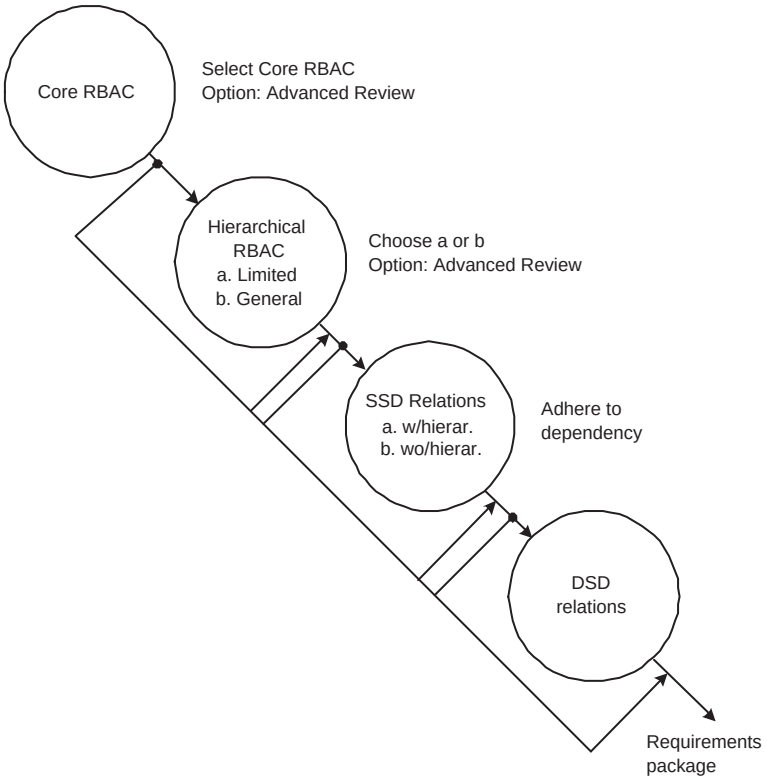

Figure 7: Functional Components of the RBAC standard [6]

Basic idea. The central terms in RBAC are *user*, *role*, and *permission* (see Figure 8). Therefore, we have sets *USERS*, *ROLES*, and *PERMS*. For elements of these sets, we have two assignment relations $UA \subseteq USERS \times ROLES$ and $PA \subseteq PERMS \times ROLES$. The *user assignment UA* defines a relation between users and roles, whereas the *permission assignment PA* defines the relation between roles and permissions. Both relations are many-to-many. The set of all permissions is obtained by the combination of *operations* and *objects*, i.e., $PERMS = OPS \times OBS$,

where *OPS* and *OBS* are the corresponding sets. Types of operations depend on the type of system which is considered. In access control terminology, an object is an entity which contains or receives information, e.g., an object may be a file or some exhaustible system resources. If a user $u \in$ *USERS* changes to a new user category leaving his old role r_{old} then he is simply assigned to a new role r_{new} by getting the permissions of r_{new} and losing the permissions of r_{old}. RBAC facilitates security management, makes it more efficient, and reduces costs.

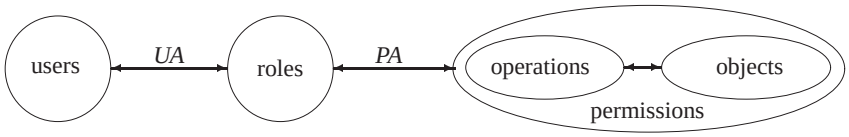

Figure 8: Core RBAC model

Role hierarchies are thought to be one of the most desirable features in RBAC. They are very useful when overlapping capabilities of different roles result in common permissions because they allow to avoid repeated permission-role assignments. This allows to gain efficiency, e.g., when a large number of users is authorized for some general permissions. Role hierarchies $RH \subseteq ROLES \times ROLES$ are constructed via inheritance relations between roles, i.e., by the introduction of senior and junior relations between roles $r_1 \succeq r_2$ in such a way that the senior role r_1 inherits permissions of the junior role r_2.

Role activation and sessions. In a role hierarchy, a senior role inherits the permissions from the junior role. But a user does not necessarily have to act in the most senior role(s) he is authorized for. Actual permissions for a user are not immediately given by evaluating the permission assignment of his most senior role(s); these roles can remain dormant. Instead, actual permissions depend on the roles which are activated. A user may decide which roles to activate. Sessions are defined over phases in which users keep roles activated. A user's session is associated with one or many roles.

Principle of least privilege. The principle of least privilege requires that a user should not obtain more permissions than actually necessary to perform his task, i.e., the user may have different permissions at different times depending on the roles which are actually activated. As a consequence, permissions are revoked at the end of sessions.

Static separation of duty. In some security policies, there may arise a conflict of interest when users get some user-role assignments simultaneously. The idea of static separation of duties is to enforce some constraints in order to prevent mutually exclusive roles. In the presence of a role hierarchy, inheritance relations have to be respected in order to avoid infringing these constraints.

Dynamic separation of duty. The goal of dynamic separation of duty is – similar to static separation of duty – to limit permissions for users by constraints in order to avoid potential conflicts. The difference here is that these constraints define which roles are not allowed to be activated simultaneously, i.e., the constraints focus on activation of roles. Roles obeying these constraints can be activated when they are required independently, but simultaneous activation is refused.

Further relevant properties. RBAC is assumed to be policy-neutral. This means that RBAC

provides a flexible means to deal with arbitrary security policies. It is highly desirable to have a common means to express a huge range of security policies.

4.3.4 Context-dependent Access Control

Many contributions to the extension of the RBAC mechanism have been made. Within SicAri we propose to extend the RBAC mechanism with respect to consideration of additional parameters within the access control decision [18]. An example for such a parameter is the identification method used to log-in to the system. Access control is strongly related to the question of user identification. There are several techniques used in practice, e.g., challenge & response protocols based on digital signatures and public key X.509 certificates, passwords, and even cookies. The decision, which of these identification mechanisms has to be applied, eventually depends on the protection goals in the corresponding application and the desired security level. High-level protection usually requires stronger mechanisms. Furthermore, there are possibly desired functionalities that can only be achieved with specific identification mechanisms. If a system provides different options for authentication against the system, the identification mechanism chosen by a user should be considered within the access control decision. That is, the user may have permission for more sensitive operations if he used a stronger identification mechanism. The access control component of the SicAri platform will provide measures to specify access control policies including rules for additional parameters, such as the identification mechanism, and considering these within the access control decision. An approach for composing context-dependent role-hierarchies from simple role-hierarchies was proposed in [17]. This approach will be used as a basis for further development.

With respect to consideration of dynamic context parameters within the access control decision, various SicAri platform components are involved:

a) The *context manager* (see Section 5.4), which stores and manages all kind of context information. This information may be provided by any SicAri component (such as the authentication manager for the user's identification mechanism), or by sensor modules (such as a sensor for the current date and time).

b) Any *basic and application service* (see Sections 5 and 6) for providing context information, e.g., the authentication manager for the user's identification mechanism, or a data retrieval service may provide information about what data has been provided to which user within the current session.

c) The *policy enforcement component* (see Section 5.1) which is responsible for collecting all necessary context information and submitting the check permission request to the policy decision component.

d) The *policy decision component* (see Section 5.2) which is responsible for evaluating the check permission request by considering all information necessary to make an access control decision, in particular the context information included in the request and the context-dependent policy specification rules.

e) The *policy specification component* holding the rules for access control decisions. The policy specification has to provide means of expression for context-dependent permissions. The development of this component is objective of a subsequent SicAri work package. Therefore, only minor information about the component can be currently provided.

Details of these components and its contribution to the realisation of context-dependent access control will be given in the respective sections.

4.3.5 Reference Monitor Approach

The term *policy enforcement* is often used in the context of access control, since most systems make use of security policies for specification of access permissions. In order to ensure that all requests from subjects (i.e., users and components) to objects (i.e., services and other resources) are evaluated with respect to whether the necessary permissions are granted by the underlying security policy, the platform must ensure that all access to sensitive objects is provided through the platform. This concept was introduced in [1] as the *reference monitor* approach.

The reference monitor is an abstract concept, whereby all access that subjects make to objects are authorized based on the security policy [5]. When a subject attempts to perform an operation on an object, the reference monitor must perform a check, comparing the attributes of the subject with that of the object. The reference monitor is policy-neutral, i.e. it does not dictate any specific policy to be enforced. There are three fundamental implementation principles for the reference monitor [5]:

1. Completeness: It must be always invoked and impossible to bypass.
2. Isolation: It must be tamper-proof.
3. Verifiability: It must be shown to be properly implemented.

A reference monitor usually consists of a single policy decision component and one or more policy enforcement components (see below). In the SicAri platform, the policy enforcement and policy decision mechanisms are completely independent of the security policy specification. Thus, it is possible to change the behavior of the platform by updating the policy specification without having to make any changes to the policy enforcement and policy decision components in the kernel themselves.

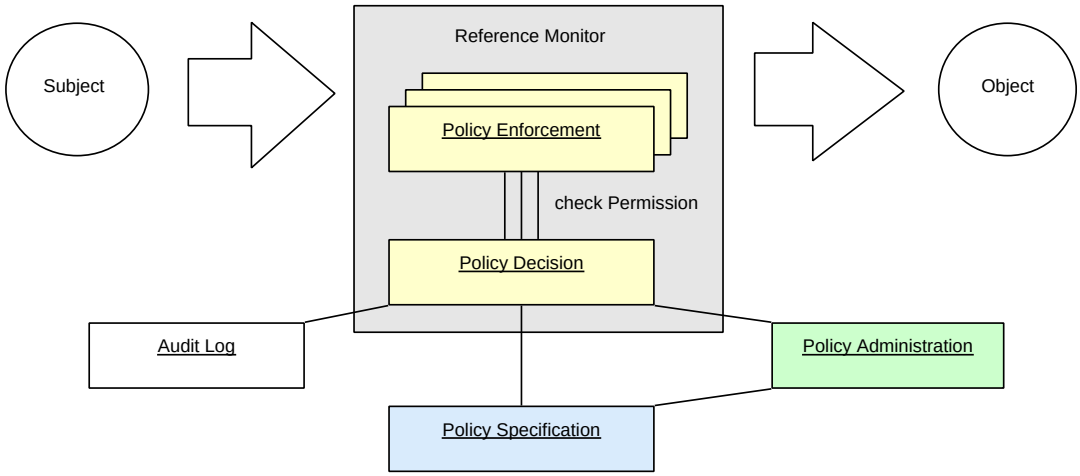

Figure 9: SicAri Reference Monitor

Figure 9 illustrates the SicAri reference monitor. Sensitive objects cannot be accessed by subjects directly. Instead the reference monitor checks whether the access is conform with the underlying security policy. The *policy enforcement* component, sends a check permission request to the *policy decision* component. The latter component is responsible for deciding whether a given request is conform to the underlying security policy. The policy decision component informs the policy enforcement component about the result, while the policy enforcement component implements this decision by either granting access to the desired object or not.

By splitting up the policy module into the policy enforcement component and the policy decision component, the SicAri architecture may be extended to support additional policy enforcement components for other kinds of application, while reusing the evaluation logic of the policy decision component.

The details about the both modules inside the reference monitor give Sections 5.1 (policy enforcement) and 5.2 (policy decision).

Figure 9 includes additional components which are related to the reference monitor. The *audit log* component is responsible for logging all requests and the corresponding decisions made by the policy decision component. As stated in [11], audit logs are important for system security. The *policy specification* component contains the machine-readable representation of the underlying security policy. Policies may change in time. Thus, the *policy administration* component is responsible for managing the security policy.

4.4 Platform Communication

Instances of the SicAri platform run on many computers. This section describes the interaction between the instances.

There are different kinds of instances in a typical SicAri infrastructure.

- Endusers starting the SicAri platform have the SicAri kernel and some SicAri services installed on their computer. This computer could be a desktop computer, a laptop or a PDA.
- On mobile phones and other small devices like the talking assistant it is probably not possible to run the SicAri kernel due to the restricted resources. They still can use SicAri services running on other machines.
- The third group of computers in a SicAri infrastructure are servers providing central services.

Platform communication always involves two steps as figure 10 shows. First the target service must be located. Second the target service must be invoked.

To find a service in the SicAri naming service the SicAri service must be registered before (step 0). All central services are registered at the naming service. Other services are free to register themselfs during initializing or on user request.

The choice of the communication protocol is up to the Service provider. But it is recommended to use the Simple Object Access Protocol (SOAP).

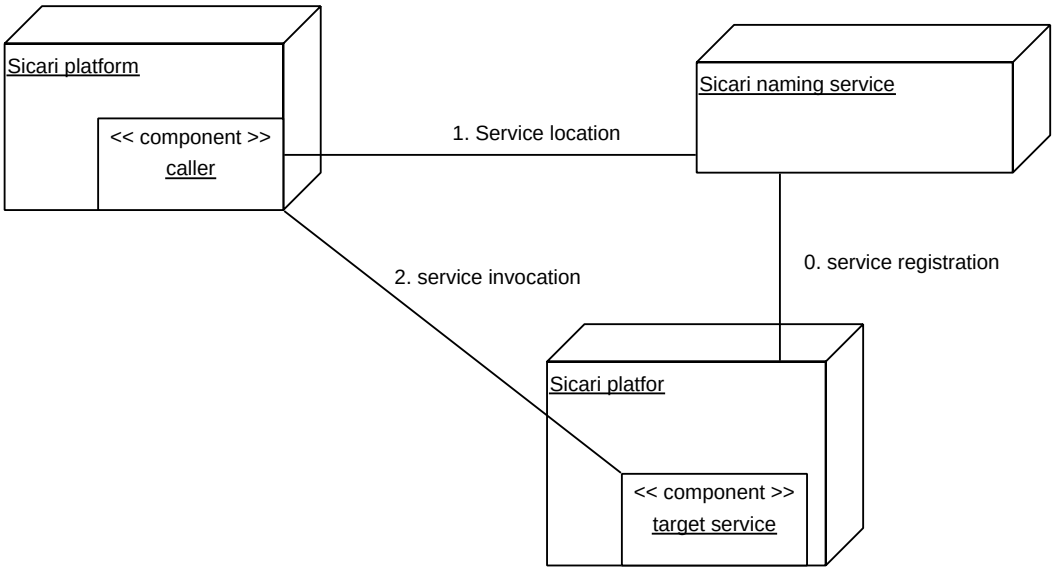

Figure 10: Communication between SicAri Platforms

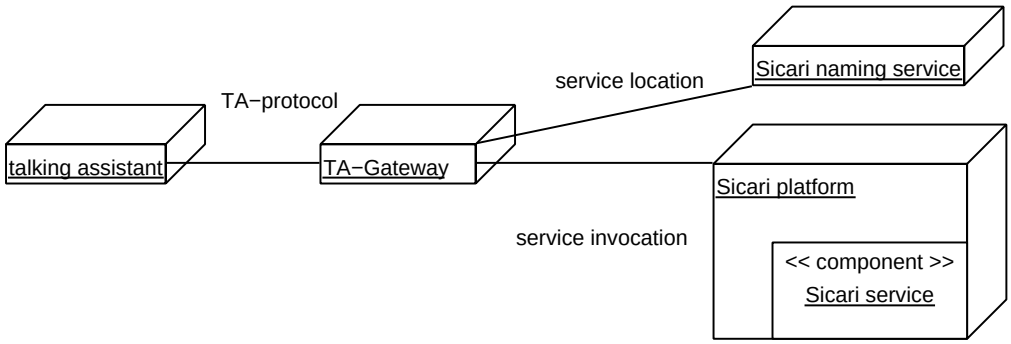

Figure 11: Communication between small devices and SicAri services

If it is not possible to use the same communication protocol for small devices, special gateways need to be installed (see figure 11). Such devices often speak optimized protocols. A gateway speaking the same protocol translates the communication to SicAri compatible protocols. The gateway must ensure secure identification of the restricted device.

5 Basic Services of the SicAri Platform

5.1 Policy Enforcement and Security Manager

The policy enforcement component and its relation to the policy decision component has already been introduced in the description of the SicAri reference monitor (see Section 4.3.5). This section gives more details about the policy enforcement component of the SicAri platform and its realisation. First, we will describe the main characteristics of this component, and second we illustrate the main use cases the policy enforcement component is involved in.

When accessing a sensitive resource (i.e. from a system view, calling a method containing a resource ID and the operation (e.g., read, write,...)) the enforcement component evaluates whether the respective user, that initiated the access request, has the necessary permissions.

In principle, there are two possibilities to activate the policy enforcement component – explicit and implicit. In the *explicit* case, each SicAri component must guarantee that each time a sensitive resource is accessed, the policy component is consulted. This approach seems to be somehow elaborate and error-prone, since the SicAri is an open extensible platform. Thus the policy enforcement call must be demanded by out-of-band mechanisms, e.g, only allowing certified SicAri modules.

In the *implicit* case, SicAri components cannot circumvent the policy enforcement component that is always running in the background. Thus, each request is evaluated by the policy enforcement component implicitly. This approach has been selected for the policy enforcement within SicAri (although the platform will also provide interfaces for explicit policy enforcement).

In order to achieve implicit involvement of the enforcement component, the platform provides a new SicAri security manager. In Java, the security manager (`java.lang.SecurityManager`) is running in the background and controls that access to sensitive resources is according to a specified policy. Java's default security manager, however, is only capable of controlling coarse-grained policies which are not applicable within SicAri. Therefor, we will replace the default security manager by a new SicAri security manager. Together with the security manager we also exchange the default policy decision and the policy specification components by new ones. The security manager delegates the policy decision to the new policy decision component (see Section 5.2), which considers the underlying policy specification. We will just outline the main aspects of the Java security manager. For further information please refer to [15, 16, 7].

Java Security Manager Overview. The security manager is an application-wide object (i.e. there is exactly one security manager per SicAri kernel) that determines whether potentially threatening operations should be allowed. The classes in the Java packages cooperate with the security manager by asking the security manager for permission to perform certain operations.

Each Java application can have its own security manager object that acts as a full-time security guard. The `SecurityManager` class in the `java.lang` package is an abstract class that provides the programming interface and partial implementation for all Java security managers.

To change the lenient behavior of the default Java security manager, SicAri creates and installs its own security manager.

The `SecurityManager` class defines many other methods used to verify other kinds of operations. For example, SecurityManager's `checkAccess()` method verifies thread accesses, and `checkPropertyAccess()` verifies access to the specified property. Each operation or group of operations has its own `checkXXX()` method.

In addition, the set of `checkXXX()` methods represent the set of operations in the Java package classes and the Java runtime that are already subject to the protection of the security manager. So, typically, your code will not have to invoke any of SecurityManager's `checkXXX()` methods – the Java package classes and the Java runtime already do this for you at a low enough level that any operation represented by a `checkXXX()` method is already protected.

However, when replacing the default security manager, some of the former security manager's `checkXXX()` methods have to be overwritten in order to tighten or modify the security policy for specific operations, or you may have to add a few of your own to put other kinds of operations under the scrutiny of the security manager.

The charm of this approach is that by replacing the security manager, we gain implicit policy enforcement, but can also integrate our own policy decision and policy specification components. The `checkXXX()` methods may also be called explicitly by any application. In this case, the policy framework can easily be extended by specific application dependent permissions as extensions of the `java.security.Permission` class. Thus, by using the Java's security manager approach we may provide both implicit and explicit policy enforcement.

Since the security manager is running in the background transparently, there is no need to adapt any applications to the new SicAri security manager. The security manager automatically calls a method `checkPermission(JavaPermission(Operation-On-Resource))`, which either returns void (in the success case) or raises an exception otherwise.

The SicAri security manager may consider context information within the permission check. By means of this approach, one may specify security policies that may deny access to file X, when the user already accessed file Y (see Section 4.3.4). In order to implement such policy rules, the system may need information about, what resources the user had access to within the current session. This can be realized by means of the context manager as follows. A SicAri component that provides access to some resource informs the context manager about the access, after the security manager granted access to that resource. For that, the context manager's API will provide a method like `accessFeedback(sessionID, userID, resourceID, actionID, timeOfAccess)`.

The realisation of policy enforcement by replacing the Java's security manager is only one possibility. The flexibility of the SicAri architecture also allows to introduce new policy enforcement mechanisms, that may also use the existing policy decision and policy specification components.

The SicAri security manager is capable of enforcing a subset of policies only – all kinds of access control policies and information flow policies. Other kinds of policies will be implemented by means of the security provisioning component (see Section 5.3).

The main interactions between the policy enforcement component and other components is illustrated in the use case representation below.

Use Cases "Access Control Enforcement"
Main Success Scenario:

1. User requests a sensitive system resource.
2. SicAri component (e.g. an SicAri application) which is responsible for data retrieval tries to access the resource (see *access Object* use case).
3. Security manager (as part of the policy enforcement module) running in the background notifies the attempt to access the resource and interrupts the retrieval until the permissions of the user to access the resource have been checked (see *check Permission* use case).
4. In order to check the permissions, two sub-use cases are called to retrieve the user's identity and context information of the user and system environment (see use cases *get Identity* and *get Context*).
5. The policy enforcement component sends a request to the policy decision component in order to evaluate whether the user has the required permissions or not.
6. The policy decision component evaluates the request with respect to the underlying security policy and makes a decision.
7. If the necessary permissions exist, the SicAri component is allowed to continue the data retrieval process (see step 2.)

Extensions:

7a. If the necessary permissions do not exist, the security manager (see step 3.) raises an exception, which causes the SicAri component to process some error handling.

Assumptions:

- Each process (and each thread) of the SicAri platform is uniquely linked to a single session ID.
- The session ID is uniquely linked to a single authenticated user, i.e., the user ID can be derived from a session ID by means of the authentication manager.

Figure 12 illustrates the use case description above.

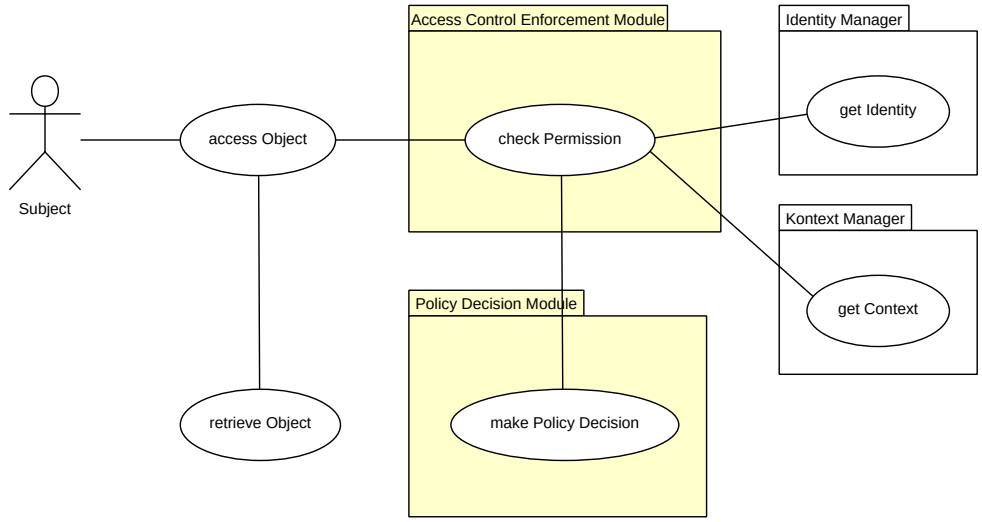

Figure 12: Use Case: Access Control Enforcement

The subsequent processes initiated by the policy enforcement component in order to make a policy decision are detailed below.

5.2 Policy Decision Component

We have extracted the decision logic from the policy enforcement component in order to be flexible enough to support different kinds of evaluation models, e.g., a simple access control model like ACLs, or a role-based model, or a more advanced role-based model, which considers context information. Additionally, by dividing up the policy component into an enforcement part and a decision part, we may support various enforcement components by reusing the same decision components.

The only objective of the decision component is to make the decision which is based on the provided information in the request and the underlying security policy. The request contains all information that is necessary to make the decision. That is, we assume that the calling enforcement component provides all necessary information in the request.

Depending on the underlying model, the request may contain at least the following information:

- user ID: the unique identifier representing a user in the SicAri platform.
- resource ID: a unique identifier representing the resource to be accessed by the user.
- operation ID: specifies the kind of access the user may perform in the resource (e.g., read, write)

More advanced models may require additional information, e.g.,

- activatedRoles: in the case that the policy may specify separation of duty constraints.
- date and time information: in the case that the policy may specify date and time constraints.

- accessed objects: in the case that the policy may specify history based constraints.

After the policy decision component came to a conclusion, it returns a reply containing the result of the permission check to the callee. Since there is currently no need to include additional information in the reply, the reply is a boolean value, which is true, if the user has the necessary permissions, and false otherwise.

The policy decision component has access to the policy specification component, which holds the security policy specification. Details of the representation of the policy specification will be given in the upcoming document about the policy specification. Here, we will use the policy specification component as an abstraction of the real representation (e.g. as an XML file or within a database model).

The policy decision component is solely used by the SicAri security manager (which is responsible for policy enforcement), since this is currently the only enforcement component. Our architectural approach however allows the development of further enforcement components that may also make use of the policy decision component.

The main use cases of the policy decision component will be illustrated now.

Use Case "Policy Decision"
Main Success Scenario:

1. The policy enforcement component (i.e. the SicAri security manager) sends a policy decision request to the policy decision component (see *check Permission* call).
2. The policy decision component, which is responsible for making decisions whether a given request is conform to the underlying security policy, receives the request from the policy enforcement component (see *make Policy Decision* use case).
3. The workflow of the policy decision component includes at least three subroutines. First, the policy request is parsed, second, the information in the request are evaluated considering underlying security policy, and third, after a policy decision is made, a reply is generated, which is a boolean in the simplest case (see use cases *parse Policy Request*, *evaluate Policy*, and *generate Policy Reply*).
4. The reply containing the result of the evaluation (e.g., true for access granted, false for access denied)

Assumptions:

- We assume that the policy decision request, which was sent by the SicAri security manager, contains all necessary information in order to make a decision, i.e. the policy decision component does not consult any SicAri component (e.g. context manager, authentication manager, identity manager) in order to retrieve additional information.

Figure 13 illustrated the use case description above.

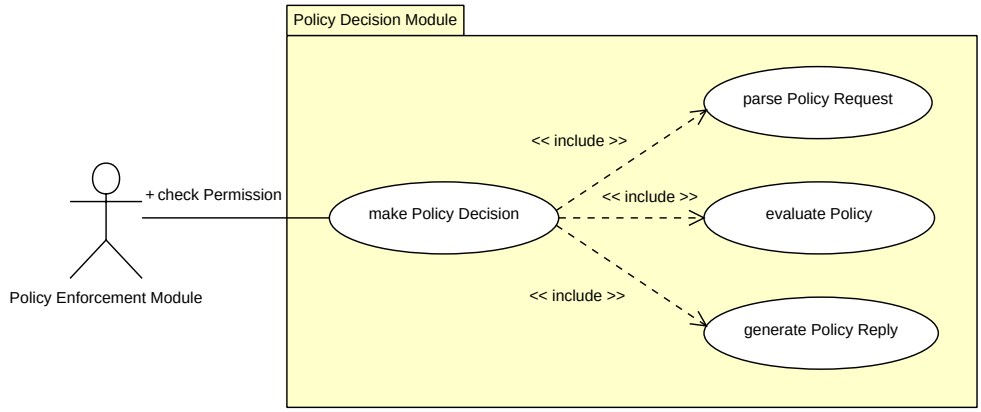

Figure 13: Use Case: Access Control Decision

5.3 Security Provisioning (Policy Enforcement for Security Mechanisms)

The goal of SicAri platform is to enable secure ubiquitous Internet access for the user who runs his application on the top of the SicAri platform. One of the platform's advantages is that the platform's trust assumption is flexible and configurable through the platform's security policies, which depends on user's needs. The defined security policies control access rights, information flows and security mechanisms, which are the building blocks in realizing security goals.

Figure 14 depicts the typical use cases of the security mechanisms.

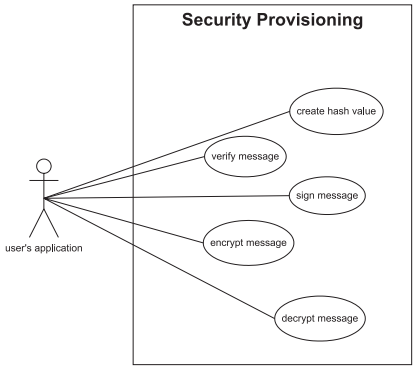

Figure 14: Use case: Security Provisioning

In this section we mainly focus on the security mechanisms that play an important role in fulfilling the security objectives. We identify the platform's security mechanisms as:

- symmetric cryptography algorithm
- asymmetric cryptography algorithm
- signature algorithm
- hash function
- Key generation

Implementation

The identified security mechanisms are non-injectively mapped onto several classes that provide these security mechanisms. To incorporate the policy-controlled mechanisms into these classes, subclasses are derived from the original classes and overwrite some of the methods. The following section enumerates these Java classes and also defines the controlled parameters and their overwritten methods.

- `KeyPairGenerator` (package: `java.security`)
  This class generates a key pair that is used by the asymmetric cryptography algorithm. The security mechanism policy controls the following parameters:
  - Algorithm
  - Key pair length

  The following method(s) will be overwritten:
  - `genKeyPair()`
    Prior to the execution, a parameters check will be performed, whether the parameters conform to the security mechanism policy. An exception will be thrown if a parameter does not conform to the policy.
  - `generateKeyPair()`
    Prior to the execution, a parameters check will be performed, whether the parameters conform to the security mechanism policy. An exception will be thrown if a parameter does not conform to the policy.
- `MessageDigest` (package: `java.security`)
  This class generates the hash value of a given byte array. The security mechanism policy controls the following parameters:
  - Algorithm

  The following method(s) will be overwritten:
  - `digest()`
    Prior to the execution, a parameters check will be performed, whether the parameters conform the security mechanism policy. An exception will be thrown if a parameter does not conform to the policy.
- `Signature` (package: `java.security`)
  This class creates and verifies signature. The security mechanism policy controls the following parameters:
  - Algorithm
  - Key pair length

  The following method(s) will be overwritten:
  - `sign()`
    Prior to the execution, a parameters check will be performed, whether the parameters conform the security mechanism policy. An exception will be thrown if a parameter does not conform to the policy.

- `Cipher` (package: `javax.crypto`)
  The Cipher class performs encryption and decryption mechanisms for both asymmetric and symmetric cryptography algorithms. This class takes the performed algorithm as one of its parameter. The security mechanism policy controls the following parameters:

  – Algorithm
  – Key length
  – Key type (symmetric or asymmetric)

  The following method(s) will be overwritten:

  – `doFinal()`
    Prior to the execution, a parameters check will be performed, whether the parameters conform the security mechanism policy. An exception will be thrown if a parameter does not conform to the policy.

- `KeyGenerator` (package: `javax.crypto`)
  This class generates a secret key that is used by the symmetric cryptography algorithm. The security mechanism policy controls the following parameters:

  – Algorithm
  – Key length

  The following method(s) will be overwritten:

  – `generateKey()`
    Prior to the execution, a parameters check will be performed, whether the parameters conform to the security mechanism policy. An exception will be thrown if a parameter does not conform to the policy.

- `Mac` (package: `javax.crypto`)
  The "Mac" class perform the Message Authentication Code algorithm based on the provided hash algorithm. The security mechanism policy controls the following parameters:

  – Algorithm
  – Key Length

  The following method(s) will be overwritten:

  – `doFinal()`
    Prior to the execution, a parameters check will be performed, whether the parameters conform the security mechanism policy. An exception will be thrown if a parameter does not conform to the policy.

Each mechanism class subscribes its policy from the `MechanismPolicyPublisher` class. This is done by calling `subscribe()` method. Therefore, each policy change can be immediately propagated to the mechanism. The propagated policy is wrapped into several classes. These are:

- `EncryptionPolicy`
- `SignaturePolicy`
- `HashMacPolicy`

Initialisation

During initialisation, the following tasks should be performed:

- A self test should be performed.

Failed to perform one of the tasks, the service should throw an Exception.

Shutting down

During the shut down process, the following tasks should be performed:

- All subscriber should be released.
- All mechanisms should not perform its task. Shutting down the Security Provisioning service while one of its mechanism is still running, should throw an Exception.

Failed to perform one of the tasks, the service should throw an Exception.

5.4 Context Manager

As the main precondition for context-aware activity support, the context manager is responsible for delivery of current information about the different entities within a service environment (as there are users, services, devices, etc.) as well as the representation of the physical surrounding. The first group of information can be described with more or less unfrequently changing values, contrary to the representation of the physical surrounding which is generally based on up-to-date sensory data. Senors modules provide this up-to-date information in an active resp. passive way (see Section 6.1).

Within the SicAri platform the context manager has various tasks. Tempory data as the current session context of a user is needed for workflow based policy decisions for example, and is stored until the session gets invalid or is terminated by the user. More static user data is directly associated with the user ID instead of the session ID and available for a longer period of time. Further the context manager has to provide an interface to the persistency service to export the database to or import the database from a persistent storage: This is very important in the case the context manager is shut down and restarted again.

The context manager stores and subsequently provides private and sensible information about entities within the SicAri infrastructure and registered users to other users and/or SicAri services. Therefore, access to the database has to be restricted by means of the system wide access control policy.

According to these requirements, the context manager is designed as hierarchical database storing key/value-pairs, whereas the key is the path from the root to a specific node in the hierarchical namespace of the database. The value either describes a certain property identified by this node in the database tree, or is a pointer to a passive sensor module which is triggered for up-to-date information upon each request. The database is partitioned into several sub hierarchies, as there are sub hierarchies for sessions, users, devices, etc.. A special sub hierarchy contains default values of defined properties which are subsequently inherited by other nodes

in the tree: This feature e.g. allows the definition of default values describing the abilities of a mobile device, which can be overwritten by current values within the requested sub hierarchy.

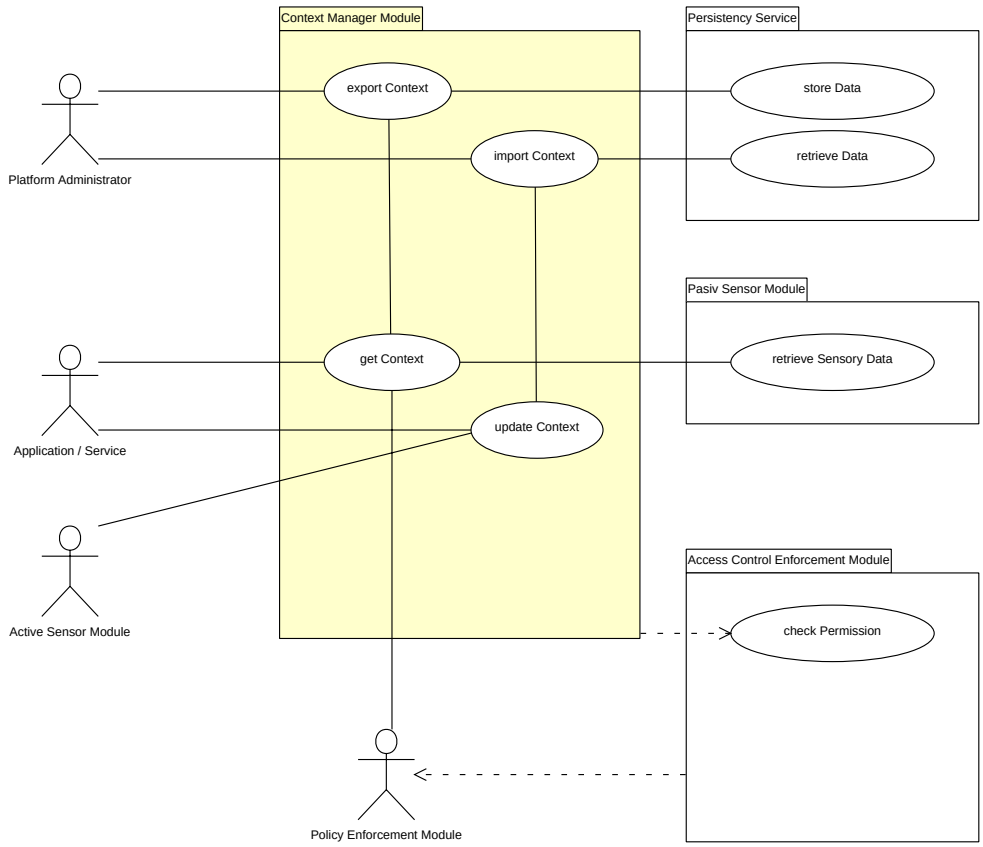

Figure 15: Use Case: Context Manager

The complete database shall be managed by the platform administrator, whereas some users of services might only receive a sub view of the database, dedicated to their access rights.

Figure 15 illustrates the different use cases of the context manager.

Use Cases "Context Manager"
Main Success Scenario:

1. Applications and services use the context database to store corresponding context information. Further, user context data can be requested according to the current access control policy of the acting user.
2. Active and passive sensor modules either alter context data automatically, or return current sensory data upon request.
3. The access control enforcement module is triggered to check the right of a user to access certain context values. On the other hand, the module acts as client if context data is needed to check workflow based policy constraints.

Extensions:

4. When the SicAri platform is started, the context database is initialized on behalf of the platform administrator.
5. Besides importing the database content from a persistent storage the administrator is able to export snapshots of the current database state back to the persistent storage. In case of platform shut down, this can be initiated automatically.

Assumptions:

- Every entity which accesses the context manager acts on behalf of a user. The administrator of the local platform is a special user within the SicAri infrastructure.
- To avoid deadlocks resp. endless loops caused by the context manager enforcing an access policy through the access control enforcement module, which in turn requests data from the context manager, the access control enforcement module has to act on behalf of special user or with privileged access rights. when this enforcement module is handled

When analysing the usecases, one cycle of module associations can be detected: As mentioned above, access to the context database should be restricted, that means access has to be controlled by the access control enforcement module. Since the policy enforcement module might access the context manager in turn to request the current user's context for workflow-based policy decisions, it is essential circumvent any dead locks, which could be generated in this scenario.

5.5 Authentication Manager

Authentication process assures the binding between the identity of subject/object and its properties. In our context, the authentication process asserts the binding between the logged user and its claimed identity. Therefore, the authentication process includes the session ID to trace the active and authenticated users.

This service works tightly together with the other two services, **Identity Manager** and **Key Store** to provide the authenticity of the subject within the SicAri platform. In SicAri platform, an active subject is authenticated, if the subject has proved its ownership of the alleged Identity stored by the **Identity Manager**. Meanwhile, the Key Store keeps his/its private key and is responsible for doing the cryptographic processes. The interaction process between the services is depicted in figure 31 and the deployment view is described in figure 16

The Authentication Manager performs the authentication mechanisms for the following subjects/entities:

- User (User name and password based authentication)
  The user gives his name as identity and authenticate himself using a password. This password is used to access the private key which is encrypted in keystore. A Challenge-Response mechanism is used to authenticate the binding between user name and X.509 certificate. The challenge-response mechanism should take place in the Authentication Manager.

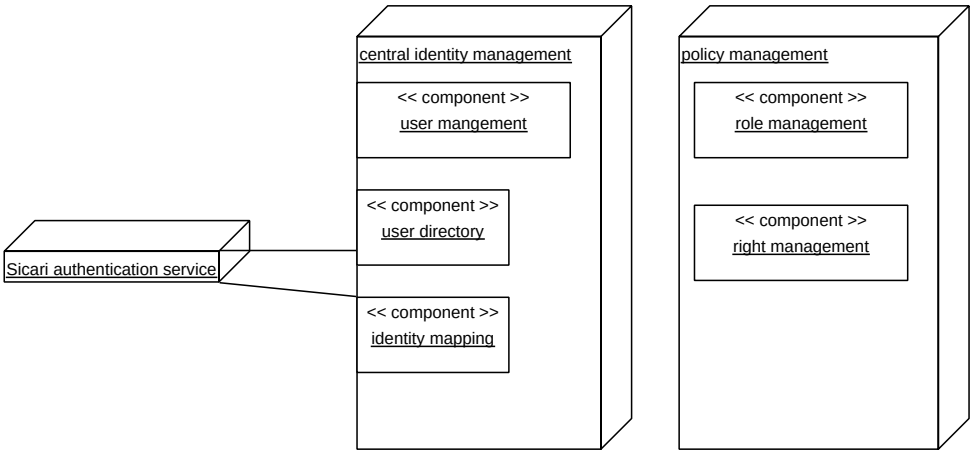

Figure 16: The deployment view of components involved in authentication process

- User (Token and PIN based authentication)
  The user initializes his token and authenticate himself using PIN. The PIN is used to access the private key which resides in the token. A Challenge-Response mechanism is used to authenticate the binding between user token and X.509 certificate. The challenge-response mechanism should take place in the token.
- User (Private-key based authentication)
  The users accessing SicAri services remotely by means of a gateway platform authenticates himself through a Challenge-Response mechanism using a locally stored private-key, which is bound to its certificate. The challenge-response mechanism should take place in the token.

The binding to the certificate assures that the claimed user is certified by an authorized instance.

Along with the authentication task, the service should also perform the following tasks:

- Perform challenge-response mechanism with the help from **Key Store** service
- Retrieve the verified certificate from **Identity Manager** service
- Store the binding between user name and certificate, and assign a session ID to it
- Receive request for signing and encrypting on behalf of an user

The general authentication process is depicted in Figure 31.

Figure 17 shows the use-case of the authentication service.

Implementation

The Authentication Manager service is implemented as **central service** as well as a **local service**. The central service provides the following service to the user or other services:

- Management of user authentication

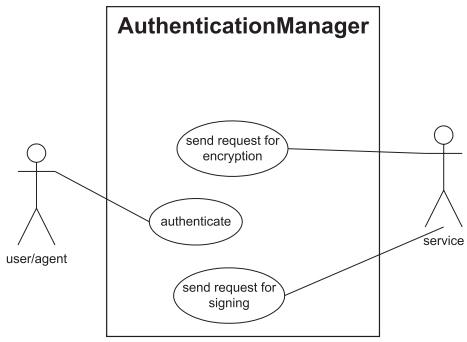

Figure 17: Use Case: Authentication Manager

- Retrieve the verified certificate
- Store the session ID

On the platform side, the local service has the following tasks:

- to act as a stub to enable the remote access to the central service
- to cache the assertion about user's authentication information stored in the central service
- to perform the challenge-response mechanism
- to perform the signing and encrypting on behalf of an user

The service has the following classes:

- `AuthenticationManager`
  The `AuthenticationManager` coordinates the authentication process and stores the user session ID. This class also receives request to perform signing and encryption on behalf of an user or agent. This request will be forwarded to the Key Manager.
- `Authenticator`
  This class groups the various Authenticators. During the authentication, the Authenticator should perform the following steps:
  1. Retrieve certificate for the specified user from the **Identity Manager** service.
  2. Generate random number and challenge the keystore to perform signing process on the random number with the given password.
  3. Received response is verified with the certificate.
  4. Store the binding between user ID and certificate in AuthenticationManager.
- `SmartCardAuthenticator`
  This class is used for smart-card based authentication.
- `PasswordAuthenticator`
  This class is used for password based authentication.
- `AgentAuthenticator`
  This class is used for agent authentication.

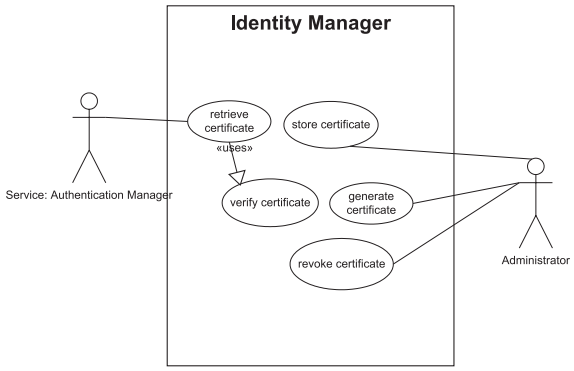

Figure 18: Identity management in SicAri

Initialisation

During initialisation, all of the the following tasks should be performed:

- self-test

Failed to perform one of the tasks, the service should throw an Exception.

Shutting down

During the shut down process, the following tasks should be performed:

- check that there is no cryptographic process running

Failed to perform one of the tasks, the service should throw an Exception.

5.6 Identity Manager

There are two views on identity management. The first view (the company view) understands identity management as central management of many users, including the aspect of user creation, role assignment and authorizing rights. The second view (the user view) understands identity management as the management of one user. In this view each user can have different user accounts. For example a user has a user account to access his company's intranet and another account to access Ebay. After a local login the correct account (or role) is used automatically through the identity management. Single-Sign-On is provided for independent services. In SicAri identity management is more understood in the first way, the company view.

Figure 18 shows the use cases of Identity Manager on SicAri Platform.

The Identity Manager has the following tasks:

- The identity Manager should guarantee that each user can be identified by each SicAri service in a secure way. To ensure this, a user is assigned to an unique certificate.

- The Identity Manager should publish the information about users in a directory service.
- The identity Manager should establish the mapping between different user's representations (between userID, email addresses, distinguished names).
- The Identity Manager should provide administrative functionalities. These are:
  - Revoke any compromised certificates
  - Issue the user's certificate
  - Import the existing user's certificate

Implementation

The Identity Manager is always implemented as **central service**.

The service has the following class(es):

- `IdentityManager`
  This is the main class of the Identity Manager service, it is implemented as singleton class. It also provides interface methods to the **Authentication Manager** service and contains the main logic for certificate retrieval and validation.
- `CertificateValidator`
  This class proves the certificate validity and its certificate chain upon request of `IdentityManager`. It is recommended to use the existing class provided by the Java Cryptographic Extension.
- `LDAPInterface`
  This class provides the LDAP interface to the `IdentityManager`.
- `AdminInterface`
  This class provides the interface regarding the administrative tasks.

Initialisation

During initialisation, all of the the following tasks should be performed:

- self-test
- initialise the LDAP database
- connect to the LDAP database

Failed to perform one of the tasks, the service should throw an Exception.

Shutting down

During the shut down process, the following tasks should be performed:

- disconnect the LDAP database
- perform the shutdown to LDAP database

Failed to perform one of the tasks, the service should throw an Exception.

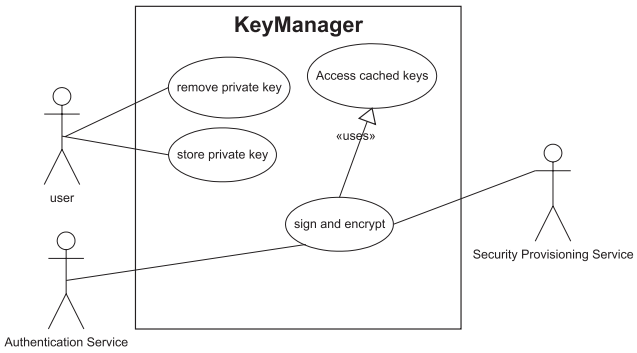

Figure 19: Use Case: Key Manager

5.7 Key Manager

The key manager has the following functionalities:

- provide encrypted private key storage
- interact with the external key storage, such as smart card, to perform the cryptographic operations
- provide key caching for single-sign-on mechanism (only available for the `Authentication Manager`)[1]
- perform cryptographic operations that take private keys as they parameters, such as: signing and asymmetric encryption (only available to the `Authentication Manager`)[2]

The following Figure 19 illustrates the use-case of the Key Manager service.

Implementation

The Key Manager is only implemented as local service, and is intended to run locally on the platform.

We enumerate again the keystore's functionalities and describe the implementations:

- encrypted private key storage
  We divide this into two cases: internal and external key storage.
  - internal key storage
    The internal key storage uses the `KeyStore` class to store private keys. This class provides an Password Based Encryption mechanism to protect stored private keys. The `KeyStore` class uses a locally stored file to save all of the private keys. The

[1] It is important to limit this operation only to the `Authentication Manager`, because the `Key Manager` does not know any information about user ID and session ID, which can allow an arbitrary instance to perform any sensitive cryptographic operations on behalf of any arbitrary user.

[2] see also note 1

user should also be given possibility to store and remove his keys into the local file. To support the single-sign-on mechanism, in each successful authentication the unsealed private keys are stored in the cache.

  - external key storage
    In case of external key storage, only the smart card or the token stores the private keys. The `KeyManager` caches only the PIN or the password.

  Both internal and external key storage protect the private keys with the PIN or password.

- supports Single-Sign-On for Authentication service
  This functionality is realized by caching the unencrypted private key's reference or caching the user's PIN within its caching period. Prior each access to the key, a check for caching period is performed.

- Connection to the external key storage
  This connection is intermediated by the Java library that comes from the smart card vendor. The actual cryptographic processes take place in the smart card itself, so that the private key never leaves the smart card.

- Perform the signing and asymmetric encryption operations
  Because the private keys should never leave the key storage, all cryptographic operations that use private keys will be performed on the keystore or on the token. This includes Challenge-Response mechanism, signature.

Initialisation

During initialisation, all of the the following tasks should be performed:

- self-test
- only for internal key storage: load the file containing the sealed private keys.
- only for external key storage: initialize the external device.

Failed to perform one of the tasks, the service should throw an Exception.

Shutting down

During the shut down process, the following tasks should be performed:

- make sure that no cryptographic processes are running.
- only for the internal key storage: save/close the file containing the sealed private keys.

Failed to perform one of the tasks, the service should throw an Exception.

5.8 Persistency Service

The SicAri platform provides a generic way to make data which is required by some component persistent. The *persistency service* provides measures for storing and retrieving any kind of data for the convenience of other SicAri services or applications.

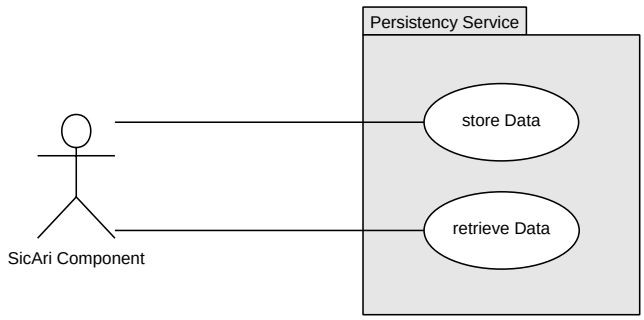

Figure 20: Use Case: Persistency Service

The use case in Figure 20 illustrates the functions of the persistency service. It basically provides two functions:

1. Store data, which takes some data to be made persistent, and
2. Retrieve data, which returns the data stored before.

Of course, the persistency service must guarantee that authorized components may access the respective data. This can be realized by configuring the underlying security policy accordingly.

Further, data volumes for special access use cases are supported. For example, audit logs can be additionally secured, if written into data volumes with the access right *write once / no rewrite*.

6 Application Services of the SicAri Platform

6.1 Sensor Modules

Sensors deliver environmental data as input to the SicAri context manager (see Section 5.4). Thereby, each sensor is responsible for a certain type of information. Real world sensory data (as there are the indoor/outdoor location/orientation of the user, or the current date/time for example) as well as sensory data from within the SicAri infrastructure (as there are user envents in the context of resource access feedback for example) is of interest.

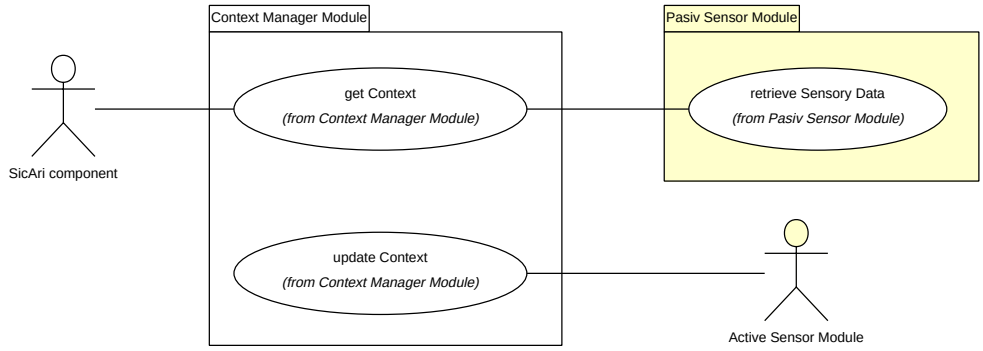

Figure 21: Use Case: Sensor Modules

Thus, sensor modules are services, which extend the context manager as illustrated in Figure 21. Two types of sensor modules can be distiguished: *Active sensor modules* automatically keep track of the sensory data and update the according nodes within the context manager hierarchy in certain intervals which are typical for the type of information acquired. Generally these sensor modules either run as `Thread` or are activated by events. *Passive sensor modules* are associated to a specific node in the context manager hierarchy and triggered by the context manager when the content of this node is requested by another service. When triggered, the sensor module acquires the information and returns in a synchronous manner. Passive sensor modules have to implement the interface `SensorModule` (see Figure 29).

6.2 Cryptographic primitives

It can not be guaranteed that todays cryptographic algorithms for encryption and signature are secure for all time. A provider architecture allows to exchange cryptographic services without reprogramming the application. In the SicAri project some research is done on new cryptographic algorithms and hardware accelerated implementations of existing algorithms. The developed algorithms must all fit in the same provider architecture.

There already exist a standard provider model in the Java world. The Java cryptographic architecture (JCA) builds a provider architecture for signature algorithms, and the Java cryptographics extension (JCE) extends this architecture for encryption algorithms.

SicAri application should use JCA/JCE for all signature and encryption operations. Cryptographic primitives developed in SicAri should provide a JCA/JCE interface.

6.3 Communication Protocols

SicAri makes no restricions in the use of communication protocols. Services may use standardized as well as proprietary protocols for communication with external entities. Of corse, the actual selected protocols should not compromise the architecture's security goals. To assure this the SicAri working group *Protocol Engineering* evaluates and suggests protocols for inter-platform communication, peer-to-peer applications, agent interaction, negotiation, service discovery, etc.

6.4 Web Services

An important task of the SicAri kernel is service management/discovery. This is done locally by the `Environment` (see Section 4.2.2). For service discovery in a distributed environment, i.e. the SicAri infrastructure, another mechanism is needed to locate and invoke services on remote SicAri platform instances. As described in Section 4.4, a naming service and a dedicated communication protocol are components of such a framework.

The work package AP MW 1 of the SicAri project proposal covers the implementation of an interoperability layer between SicAri and non-SicAri components using web services. Within this work package a framework will be implemented that can be used to fulfill the requirements of service discovery and service interaction in a distributed environment as descripbed above.

Mainly the three following XML-based standards are covered by the web service specification:

UDDI: *Universal Description, Discovery, and Integration (UDDI)* defines a platform-independent, XML-based registry to list services on the Internet. It is designed to be interrogated by SOAP messages and to provide access to WSDL documents describing the protocol bindings and message formats required to interact with the web services listed in its directory.

WSDL: The *Web Service Description Language (WSDL)* is an XML format published for describing web services. WSDL describes the public interface to the web service. This is an XML-based service description on how to communicate using the web service; namely the protocol bindings and message formats required to interact with the web services listed in its directory. The supported operations and messages are described abstractly, and then bound to a concrete network protocol and message format.

SOAP: *Simple Object Access Protocol (SOAP)* is a light-weight protocol for exchanging messages between computer software, typically in the form of software components. The word object implies that the use should adhere to the object-oriented programming programming paradigm. SOAP is an extensible and decentralized framework that can work over multiple computer network protocol stacks. Remote procedure calls can be modeled as an interaction of several SOAP messages. SOAP is one of the enabling protocols for web services

In the context of SicAri we will use open source implementations of these three standards. An UDDI directory will be applied as naming service, whereas WSDL and SOAP are implicitly used for service description and service interaction.

The goal of the framework described in work package AP MW 1 is the extension of the local service manager in a transparent way for the user. Services registered in a certain hierarchy of the `Environment` shall automatically be deployed as web service and registered in the UDDI directory with an automatically generated WSDL description. Subsequently, local access to the service's JavaAPI shall automatically invoke the remote service implementation via SOAP.

Further information will be given in the corresponding report of work package.

6.5 Agent Service-Framework

The *SeMoA*[3] *(Secure Mobile Agents)* system is a security-centric middleware for mobile agents. As illustrated in Figure 22, mobile agents (displayed as △) are able to migrate from one SeMoA server to another, communicate with other agents, and interact with the host environment through local service interfaces. The SeMoA middleware provides a sophisticated security model which is able to handle a variety of security threats.

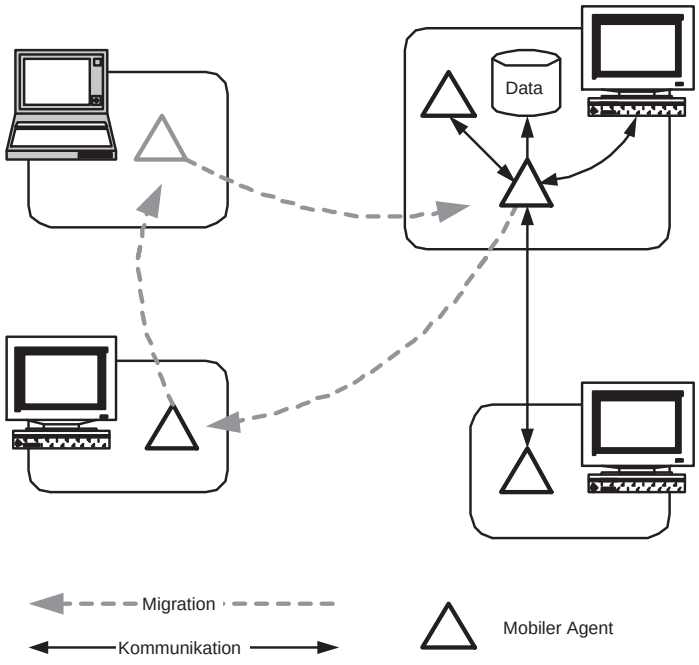

Figure 22: The SeMoA Architecture

The paradigm of migrating agents (i.e. program code coupled with its current execution state and further data) from one host to another, which act on behalf of their owner in an autonomous way, can be used for delegation purpose. The security model thereby garanties proper enforcement of security policies in a remote environment, according to the agent's owner and other agent properties.

Beside these features, the SeMoA system provides a variety of other services for distributed environments. Since this service framework for agents is deployed on a middleware, which design is similar to the SicAri platform design, it is quite easy to establish an agent server on top of a SicAri platform by loading the according service modules, if needed.

The work packages AP PE 12 and AP PE 13 of the SicAri project proposal cover two protocols, which improve the robustness and security of agent migration. More details about these protocols will be published in the according reports of the work packages. The implementations will then be available as part of the agent service framework.

[3] SeMoA: http://www.semoa.org/

7 Logical View

The logical view shows both the static and dynamic views of the system. The logical view concentrates on getting the best logical grouping of functionality into objects. The main objects of this view are:

1. Classes
2. Stereotypes
3. Packages
4. Class Diagrams
5. Relationships
6. Collaborations
7. Interactions

Objects 1–5 mainly cover static aspects, while the objects 6–7 describe dynamic aspects of the system.

7.1 Static Aspects

Class diagrams describe the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. The UML uses the term feature as a general term that covers properties and operations of a class. Packages are used to group classes with related functionality.

7.1.1 Policy Enforcement

As described in Section 5.1, SicAri replaces the Java's default security manager `java.lang.SecurityManager`. In order to provide a new security manager, SicAri provides a subclass of the `de.sicari.SecurityManager` class `de.sicari.security.SecurityManager`. The following Java pseudocode illustrates the new SicAri security manager.

```
package de.sicari.security
class SecurityManager extends java.lang.SecurityManager {
    ...

    /* policy decision component */
    private PolicyDecision decisionMaker;

    /* policy request object */
    private DecisionRequest request;
    ...

    /* sample checkPermission method */
    public void checkPermission(...) {
       decisionMaker = new SimpleAccessControl(config);
       request = new SimpleRequest();
       // fill in all necessary information into request object
       ...
       if (! (decisionMaker.makeDecision(request)))
          throw new SecurityException("Permission Denied!");
    }
    ...
}
```

The SicAri SecurityManager subclass overwrites various methods from `java.lang.SecurityManager` in order to customize the verifications and approvals needed in SicAri platform. For example, the new security manager does not evaluate the various `checkXXX` methods itself, but consults the SicAri policy decision component (see next subsection).

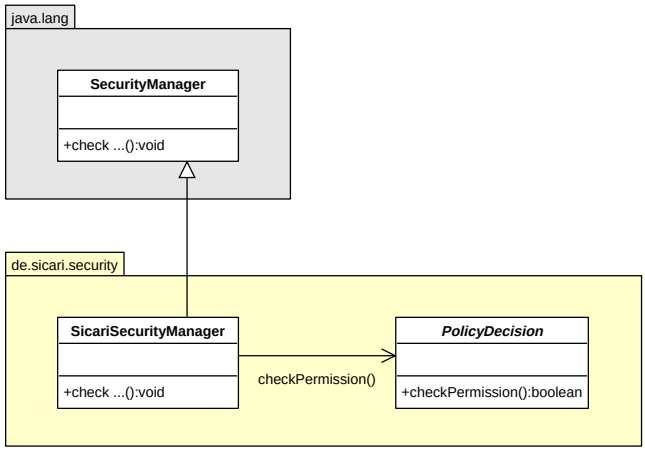

Figure 23: Class Diagram: SicAri Security Manager

All classes related to policy enforcement and decision are located in the package `de.sicari.security`. Figure 23 outlines the SicAri security manager classes and its relation to the policy decision component.

7.1.2 Policy Decision

All classes related to policy decision are also located in the package `de.sicari.security`. As described in Section 5.2, the security manager sends a decision request to the policy decision component in order to check permissions. Since SicAri will support various mechanisms for permission checking, there is an abstract class `de.sicari.security.PolicyDecision` which is implemented by the various forms of permission check mechanisms. For example, there is a simple access control mechanism `de.sicari.security.SimpleAccessControl` which provides ACL like functionality. As illustrated in Figure 24, there are also decision engines for the Core Role-based Access Control mechanisms (`de.sicari.security.CoreRBAC`) and a Context-dependent RBAC mechanism (`de.sicari.security.ContextRBAC`). Of course, each decision mechanism uses its own request class (see classes `SimpleRequest`, `RBACRequest`, and `ContextRBACRequest`) which implement the abstract class `de.sicari.security.DecisionRequest`.

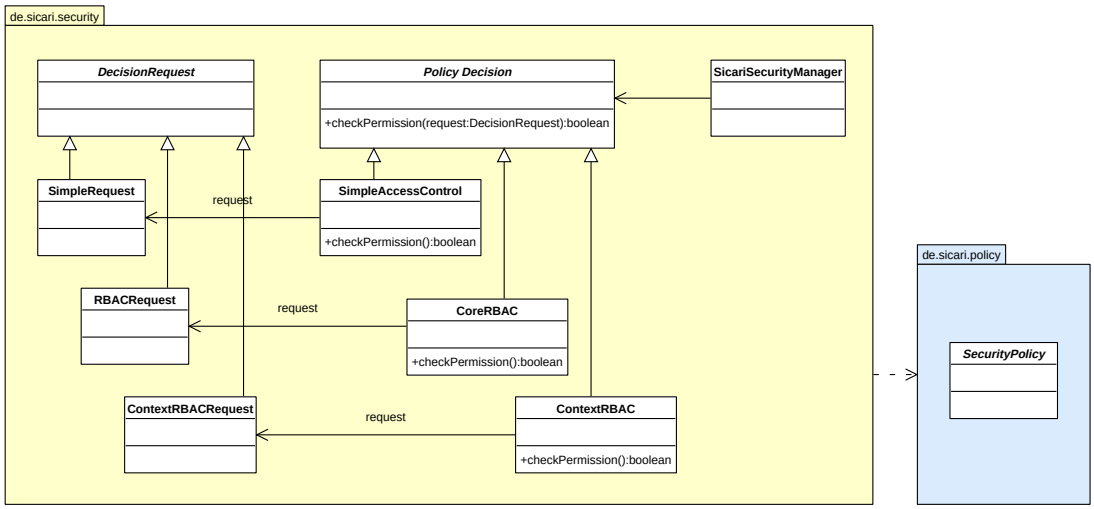

Figure 24: Class Diagram: Policy Decision Component

The picture also shows a dependency between the decision components and the policy specification package. All classes related to policy specification are located in the package `de.sicari.policy`.

Of course, there are more dependencies, for example to the administration component of the SicAri platform. This and other dependencies have been omitted for the sake of complexity.

7.1.3 Security Provisioning

The classes in Security Provisioning service are packaged in `de.sicari.mechanism`. The classes extend several classes from `java.security` and `javax.crypto` packages.

The Security Provisioning classes extend the internal java classes, which are responsible for cryptographic operations. These classes are enhanced in terms of their configuration, which is policy-controlled.

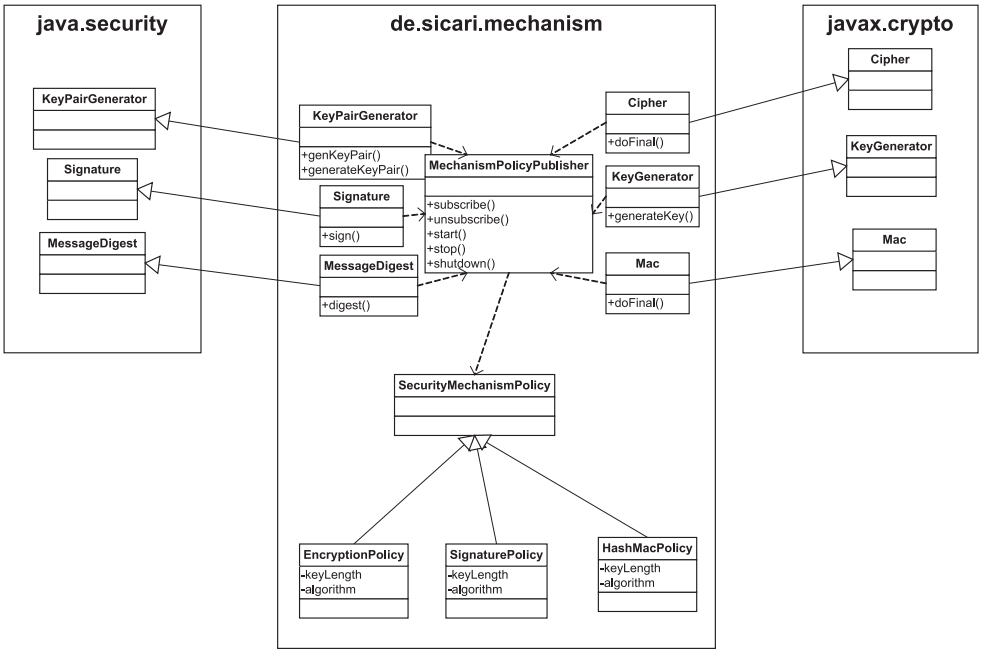

Figure 25: Class Diagram: Security Provisioning

7.1.4 Authentication Manager

The classes of Authentication Manager service in figure 26 are packaged in `de.sicari.authentication` and implemented as local service as well as central service. These classes should be implemented as central service and as a stub of local service:

- `AuthenticationManager`

These classes should be implemented as local service:

- `Authenticator`
- `SmartCardAuthenticator`
- `PasswordAuthenticator`
- `AgentAuthenticator`

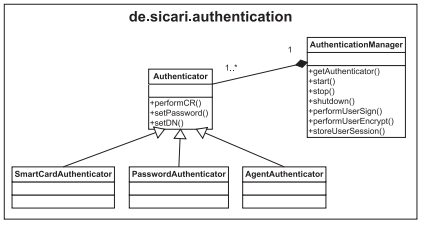

Figure 26: Class Diagram: Authentication Manager

7.1.5 Identity Manager

Because the Identity Manager service works tightly together with the Authentication Manager service, their classes are packaged in `de.sicari.authentication`

The `IdentityManager` class provides method to retrieve the user's certificate and has the following classes:

- `AdminInterface`
  This class enables the administrator to manage the stored identity.
- `LDAPInterface`
  This class provides an access to the certificate repository.
- `CertificateValidator`
  A certificate validator can be implemented in many ways. It is reasonable to separate the certificate's validation mechanism into a different class.

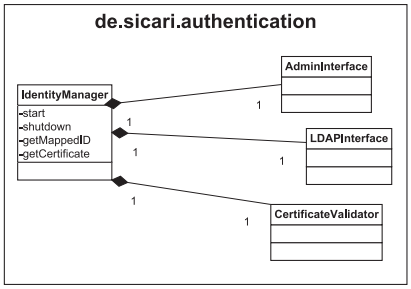

Figure 27: Class Diagram: Identity Manager

7.1.6 Key Manager

The Key Manager class has only one class, namely the `KeyManager` class. This class is also packed together in `de.sicari.authentication` with the other classes form the two previous services.

The `KeyManager` class uses the `KeyStore` class from the `java.security` package. It needs to access the keys stored in `KeyStore`.

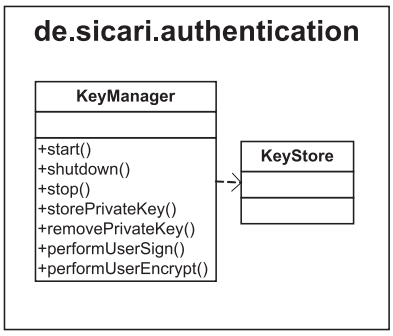

Figure 28: Class Diagram: Key Manager

7.1.7 Context Manager

The context manager consists of the class `SicariContextManager`, which uses a specific implementation `ContextDBImpl` of the interface `ContextDB` to store the context data within a hierarchically structured database. Besides the functionality of setting, getting ,and removing key/value-pairs (the key is the path within the hierarchy), this interface allows the caller to create subviews, import data from resp. export data to XML files, and to add watch points within the database to inform the caller when the corresponding entry changes.

As described in section 5.4, the `SicariContextManager` further triggers registered sensor modules `SensorModuleImpl` implementing the interface `SensorModule` to request current sensory data for specific entries wihtin the database which are marked as sensor entries. All relevant context manager classes are packed together in `de.sicari.context`.

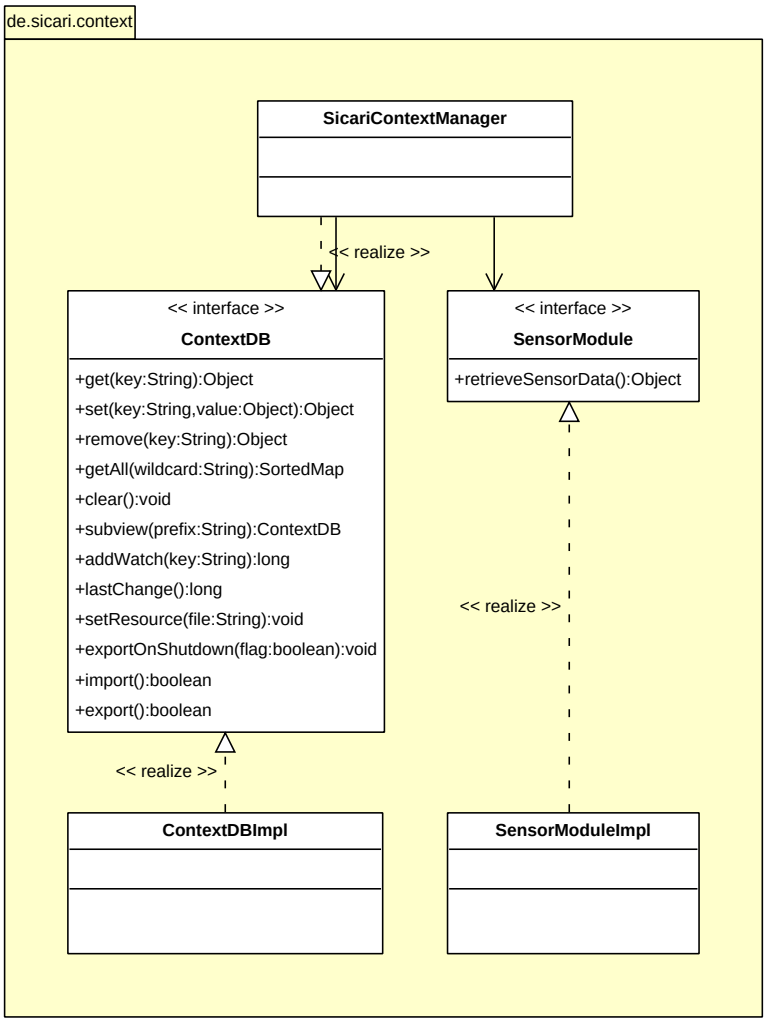

Figure 29: Class Diagramm: Context Manager

7.2 Dynamic Aspects

While sections above mainly concentrated on single platform components and static dependencies, this section focuses on interactions between platform components. Two important scenarios will be described – check permission and authentication of users against the platform.

Chronological interactions between various components are usually illustrated by UML sequence diagrams. Typically, a sequence diagram captures a single scenario. The diagram shows a number of objects and the messages that are passed between these objects. Each object is represented by a life line that runs vertically down the page and the ordering of messages by reading down the page. The messages are labeled with simplified function calls that shall help to understand the intention of the respective message.

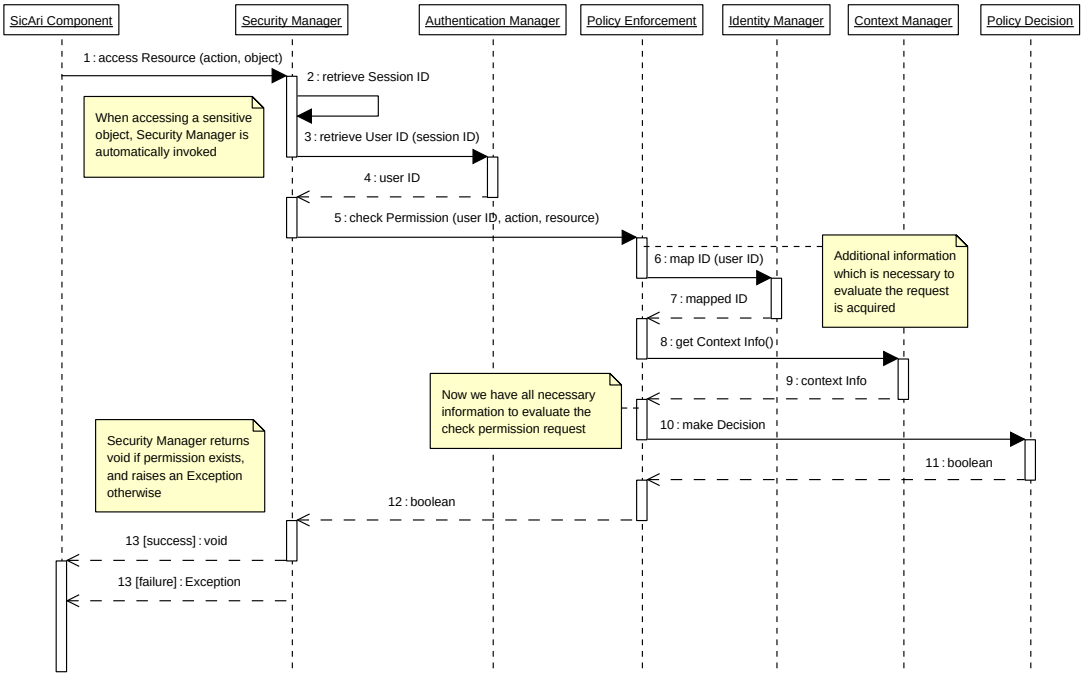

Figure 30: Sequence Diagram: Check Permission

7.2.1 Scenario: Check Permission

This scenario illustrates the interactions between the various components involved in access control. The step numbers correspond to the respective message numbers in the sequence diagram (see Figure 30), which graphically depict the step descriptions below. Consider the following scenario:

Step 1: A user working with the SicAri platform needs to access a sensitive resource (e.g. a confidential file). The SicAri component that is responsible for the data retrieval tries to access the resource on behalf of the user. The security manager running in the background notices the access attempt of the SicAri component and checks whether the necessary access permissions exist.

Step 2: The security manager identifies the thread ID of the process and the session ID of the request.

Step 3: The security manager retrieves the user ID of the requesting user by means of the session ID of the request via the authentication manager.

Step 4: The authentication manager returns the user ID.

Step 5: The security manager sends a check permission request containing user, the kind of action and the resource ID to the policy enforcement component.

Step 6: The policy enforcement component is responsible for retrieving additional information which is necessary to evaluate the request. Potentially, the policy enforcement component may need another representation of the user's id, e.g. the user's distinguished name of the public key certificate.

Step 7: The identification manager return some mapping of the user's ID.

Step 8: If the access control policy includes context-dependent policy statements, the policy enforcement components requests all necessary context information from the context manager.

Step 9: The context manager returns the context information to the policy enforcement component.

Step 10: Now, all necessary information for the evaluation of the request are available. The policy enforcement component creates a request for the policy decision component, which includes all information which is required for the evaluation of the request, and sends the request to the policy decision component.

Step 11: The policy decision component analyses the request and makes a decision whether the requesting user has sufficient permissions to access the requested resource. A boolean value representing the decision of the policy decision component is sent back to the policy enforcement component.

Step 12: The policy enforcement component forwards the boolean value to the security manager.

Step 13: The security manager running in the background translates the boolean value. It returns void, if the component acting on behalf of the user has the permission to access the resource. Otherwise the security manager raises an exception that forces the calling SicAri component to perform some exception handling.

7.2.2 Authentication

The sequence diagram in Figure 31 shows a successful authentication that is directly followed by the access from any arbitrary service.

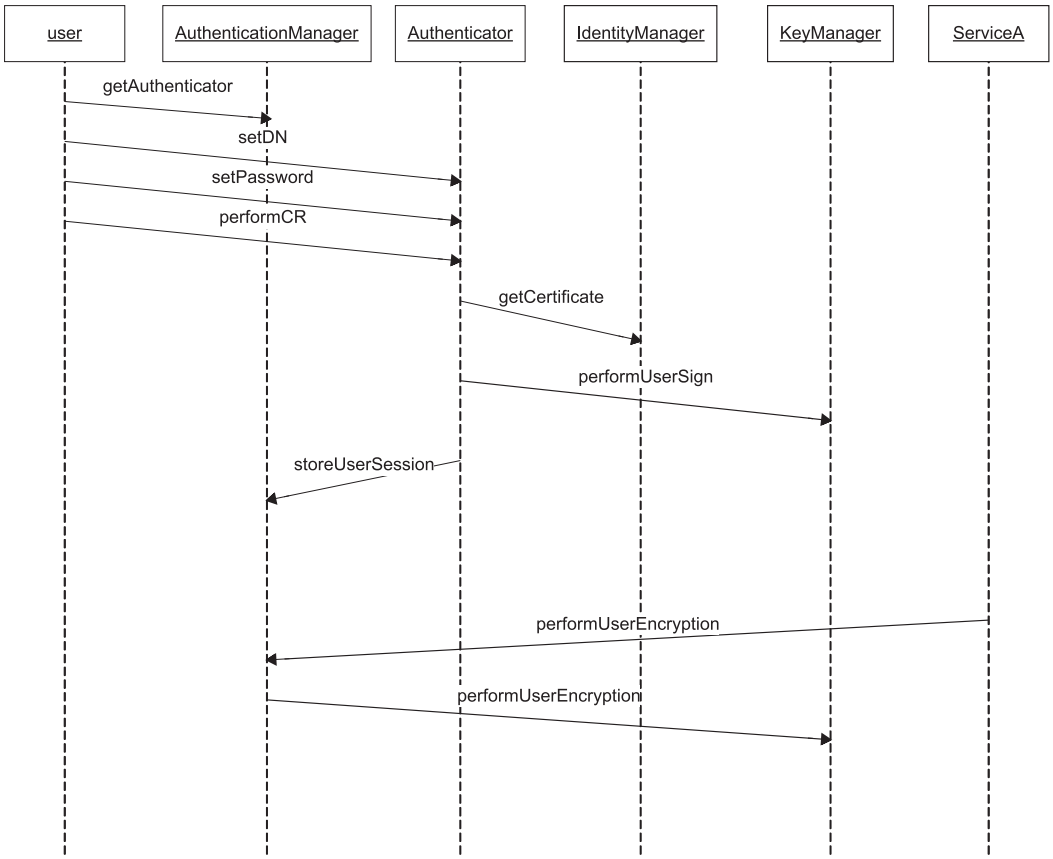

Figure 31: Class Diagram: Authentication Manager

8 Deployment View

On each SicAri platform the SicAri kernel provides a plug-in framework for modules and basic security mechanism. But SicAri is also a distributed system. This view is described below.

As you can see in Figure 32 there are some services that are deployed on dedicated systems only. These systems don't need to run a SicAri platform, whereas in any case the services are integrated into the SicAri infrastructure by means of according service interfaces available on a dedicated SicAri platform.

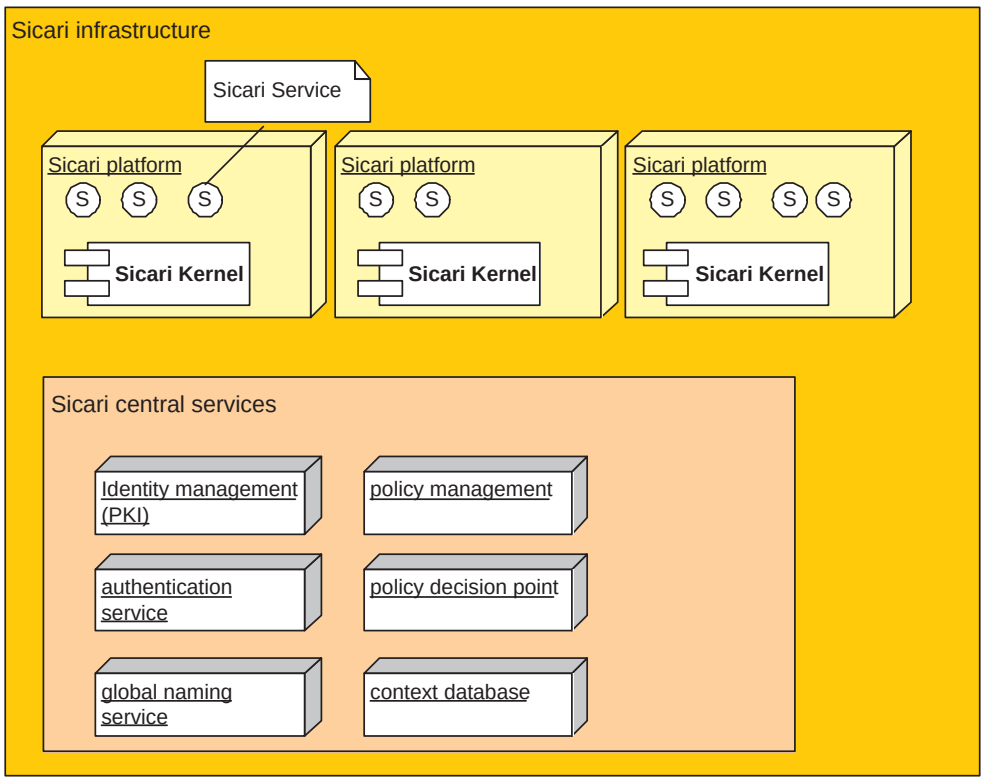

Figure 32: SicAri Deployment View

Following dedicated services are identified so far:

- SicAri identity management

  The management of identities includes the registration of new users and maintenance of existing ones. The aspect of role assignment and authorisation (giving access rights to a role, respectively assign roles to a user in the current session) is part of the policy management in SicAri.

- SicAri policy management

  While policies are enforcement locally at the platform instance, the configuration of policies is accomplished at a central place.

- SicAri policy decision point

  The local policy enforcement point aska a central policy decision point (PDC). Platform instances can either load all policy information during startup or ask the PDC every time they need a decision.

- SicAri context database

  The SicAri context database stores global context data. For example a unique session id is stored for each session of a user, and current context information is associated with this session. Another example is positional data of a SicAri platform (very interesting in scenarios with mobile devices).

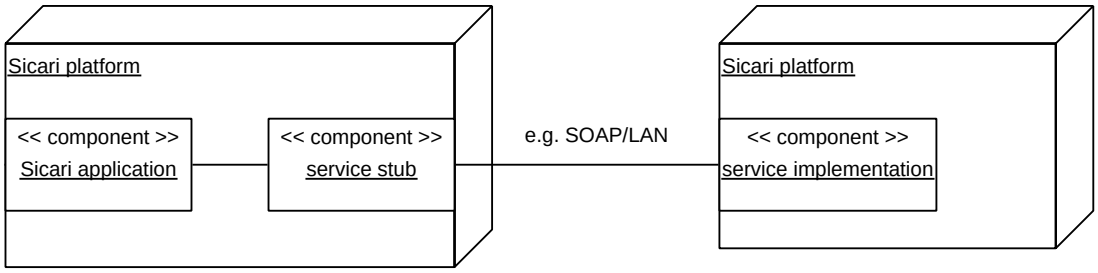

Figure 33: Accessing remote SicAri services

- SicAri naming service

  To locate central services or services on other platforms a platform can ask a central naming service. This naming service does not dictate one communication protocol.

To access server side components easily stubs are provided at each SicAri platform. SicAri application programmers can access remote services as if they were local this way. This is shown in Figure 33.

9 Programming Guidelines and Examples

This section defines the programming guidelines for SicAri, and provides developers with information about tools and example source code. Although, the following has a strong focus on the developing environment for the SicAri platform, the guides are kept as general as possible. Source code developed in the context of the SicAri project shall be stored and managed with the Concurrent Versions System CVS[4].

9.1 Code Conventions and Tools

The SicAri platform and most of its components use Java Technology. Thus, we provide *SicAri Code Conventions* [3] containing guidelines for writing Java code within the SicAri project. More precisely, this document refers to a number of existing style guides, and amends or modifies them as to reflect the particular “flavour” which is promoted.

There are several good reasons (quoted in the document) to have a common style guide. But since it’s easier to define code conventions than to actually conform to them, we additionally provide two open source tools together with SicAri codestyle conform configuration files within the CVS repository: *CheckStyle*[5] allows fine granular definition of code convention rules, which are checked automatically for given source files. In addition to syntax checking the tool supports a number of semantic programming rules which report to the user on different report levels (error, warning, info, etc.). *Jalopy*[6] is comparable to CheckStyle. It does not support the variety of semantic rules, but much more important, it automatically corrects the code style of given

[4]The SicAri CVS server is located on `cvs.sit.fraunhofer.de`. For further information contact Ruben Wolf (ruben.wolf@sit.fraunhofer.de).

[5]CheckStyle: http://checkstyle.sourceforge.net/

[6]Jalopy: http://jalopy.sourceforge.net/

source files according to the configuration. Both tools are Java-based and provide a console application as well as several PlugIns for integrated development environments (i.e. Ant, Eclipse, etc.)

When implementing a security platform, logging and testing are important aspects of the developing process. We provide a *Logging Wrapper* with a well defined Java API, which supports Sun's Logging Framework (available since JDK 1.4.0) as well as Apache's Log4J (for backward compatibility). A corresponding user guide can be found in the CVS repository [2]. For testing purpose *JUnit*[7] shall be used.

In addition to the SicAri code conventions we will provide the a *SicAri Developer's Guide* [4] containing general programming guidelines and the definition of SicAri Java APIs to use. For the time being we refer to Sun's *Security Code Guidelines* [9] and the following of this section.

9.2 The SicAri Prototype

The *SicAri Prototype* constitutes the implementation of the SicAri platform, including the SicAri launcher, the SicAri kernel, and further the basic and application services as defined in this document.

9.2.1 Directory structure in the CVS repository

```
<cvs>/sicari/prototype

  /bin               - Scripts, Configuration Files for Development Tools
  /classes           - Java Sources/Classes (*.java, *.class)
  /doc               - Documentation
    /codestyle         - SicAri-Codestyle
    /logging           - Logging Wrapper Framework
    /sicari-kernel     - SicAri-Kernel Documentation (Shell)
  /etc               - Configuration Files
  /lib               - 3rd-Party Libraries (*.jar)
    /licenses          - Licenses of 3rd-Party Libraries
  /templates         - Templates
  /test              - Java Sources/Classes for JUnit tests
  /tools             - Development Tools
    /apache-ant        - Ant 1.6.2 (http://ant.apache.org)
    /checkstyle        - CheckStyle 3.4 (http://checkstyle.sourceforge.net)
    /jalopy            - Jalopy 1.0b11 (http://jalopy.sourceforge.net)
    /junit             - JUnit 3.8.1 (http://www.junit.org)
```

9.2.2 Installation

- Java JDK (Ver >= 1.4.x) has to be installed on the computer.
- Install the various SicAri tools by unzipping the according files within the directories: `tools/*/`

[7] http://www.junit.org/

- Set the property `sicari.base.dir` within the Apache-Ant build file `bin/sicari-ant-build.xml` to the local base directory of the SicAri prototype (e.g. '<cvs>/sicari/prototype').
- Use the proper Apache-Ant script in `tools/apache-ant/apache-ant-1.6.2/bin` to run Ant. Initialize the SicAri-Prototype with the following options: `<ant-script> -f bin/sicari-ant-build.xml setup`
- Configure the single properties (if not initialized automatically) within the opened SicAri-Starter GUI, until the `Start`»-Button is activated. Then close the SicAri-Starter GUI by pressing the `X`-Button of the window frame.
- The SicAri properties will be stored in the file `<user_home>/.sicari.properties` then.

9.2.3 Apache-Ant build file

The following command gives an overview about the available Ant tasks:

```
<ant-script> -f bin/sicari-ant-build.xml -projecthelp
```

For further use of Ant, it's more convenient to define an according alias which automatically uses the right build file `bin/sicari-ant-build.xml`.

9.2.4 SicAri Launcher

The *SicAri Launcher* provides GUI-based startup and configuration of the SicAri platform with the following functionality:

- Automatically searches for Java-SDK installations.
- Automatically searches for SicAri-Prototype.
- Sets Java-Classpath (`java.class.path`).
- Sets Java-SecurityPolicy (`java.security.policy`).
- Sets Logging-Wrapper-Configuration (`DE.FhG.IGD.logging.config`).
- Sets Shell-Parameters (`sicari.base`, `sicari.etc`, `sicari.log`).
- Starts the SicAri-Kernel via SicAri-Shell.

The Java package prefix within the CVS repository is `de.sicari.kernel.starter`. Following command line paramters are available:

- **-help**
- **-nogui**
- **-f** $sicari\_start\_script$ [$script\_parameters$]
- **-rc**
- **-daemon**
- **-config** $properties\_file$
- **-D** key = $value$

9.2.5 Regular start

Change the current directory to `<cvs>/sicari/prototype`, and run the SicAri-Kernel via the following command: `java -jar sicari-launcher.jar -nogui`

9.3 The Shell

The *SicAri Shell* as part of the SicAri kernel is started automatically by the launcher. After bootstrapping basic and application services (in genral via a so-called shell script) the main task of the shell is the provision of an interaction layer for the platform administrator, which is comparable to a UNIX shell. The Java package prefix within the CVS repository is `de.sicari.kernel.shell`.

9.3.1 Command line parameters

- **-help**
- **-f** $init\_script$ $script\_options$
- **-daemon**
- **-out** $filename$
- **-err** $filename$

9.3.2 Built-in commands

- **alias** [$identifier$ [$definition$]]
- **cd** [$path$ | **all**]
- **echo** [[**-n**] $string$]
- **eval** [$variable\_name$] $op1$ $operator$ $op2$
- **exit** [$status$]

- **help** [*command*]
- **history** [(**!** | *line_number* | **-***count* | *prefix* | **?***string*) [*append*]]
- **java** *class* [*arg**]
- **permissions** [**-s**] *class**
- **print** *path_with_wildcards*
- **printenv** [**all**]
- **prompt** *command*
- **publish** *class path* [*flags*]
- **read** *variable_name*
- **retract** [**-v**] *path*
- **run** *class* [**&**]
- **security** *class*
- **setenv** *key* [*value*]
- **skip** *label*
- **source** *script*
- **threads**
- **unalias** *identifier*
- **version**
- **why** [*count* | **all**]

9.3.3 Shell Syntax

- **Piped commands:**

  command [| *command*]*

  ```
  echo "Input for the following command" | java class
  ```

- **Seperated commands**

  command [**;** *command*]*

```
echo command1; echo command2
Publish -key ${WhatIs:INGATE}/raw \; -setMax 0
```

- **Variables**

  ${*java_system_property***}**
  ${*shell_variable***}**
  ${WhatIs:*whatis_name***}**

```
setenv shell_variable test

echo ${user.dir}              /home/sicari
echo ${shell_variable}        test
echo ${WhatIs:MY_SRV}         /public/my_service
```

- **Strings**

  ${*variable_name***}**
  ‘*command_in_backticks***‘**
  "*double_quoted_string***"**
  ’*single_quoted_string***’**
  string_with_escaped_characters

```
echo a  b c           a b c
echo a\  b\  c        a  b  c
echo "a  b  c"       a  b  c
echo "’" ’"’         ’ "
echo ${PWD}           /
echo ‘echo test‘      test
```

- **while-Loops**

  while *condition***; do** *body***; done**

```
setenv  i  2

while  eval ${i} >= 0
do
    echo "i =" ${i}
    eval i ${i} - 1
done
```

- **for-Loop**

  for $variable\_name$ **in** $string\_list$**; do** $body$**; done**

```
setenv name "Jan Peters"

for name in "Uli Pinsdorf" "Peter Ebinger" "Mehrdad Jalali "
do
    echo "Local name:" ${name}
done

echo "Global name:" ${name}
```

- **if-Condition**

  if $condition$**; then** $body$**;** [**else** $body$**;**] **fi**

```
setenv  i  7

if eval ${i} > 3
then
    echo "i > 3"
else
    echo "i <= 3"
fi
```

- **case-Condition**

  case $string$ **in** [$pattern$ [| $pattern$]***)** $body$ **break;**]* **esac**

```
case ${i} in
    test | Test )
        echo ''test' oder 'Test'"
        break
    prefix*)
        echo "'prefix' with an arbitray suffix"
        break
    a?a )
        echo "prefix 'a', any character, suffix 'a'"
        break
    *)
        echo "none of the previous"
        break
 esac
```

9.3.4 Interfaces for developers

The shell provides a number of Java interfaces, which shall be used by service developers to integrate their Java classes into the SicAri platform:

`de.sicari.service.AbstractService` The abstract service class provides convenience methods as shell interface and a service dependency checker mechanism for the service developer. This class shall be used as super class for SicAri services:

```
class SicariServiceImpl extends AbstarctService
```

`de.sicari.util.Variables` Shell variables can be accessed via the variable context:

```
VariableContext vc = Variables.getContext();
```

`de.sicari.util.WhatIs` This is another convenience class, which maps a general service name to the local path within the environment. Its functionality can be used directly on the shell layer via the variable syntax `${WhatIs:`$whatis_name$`}` or within class implementations:

```
String key = WhatIs.stringValue("CERTFINDER");
```

9.4 The Environment

As the local SicAri service manager, the `Environment` allows to `publish`, `lookup`, and `retract` services. This functionality can either be used directly on the shell layer, or within a service implementation. The following example shows how access the environment within Java.

```
package de.sicari.service;

public class ServiceImpl implements Service
{
    static public void main(String argv[])
    {
        Environment env;
        Service srv;
        String key;

        env = Environment.getEnvironment();
        srv = new ServiceImpl();
        key = "/public/my_service"

        try
        {
            /* publish a service
             */
            env.publish(srv, key);

              ...

            /* lookup a service
             */
            srv = (Service)env.lookup(key);

              ...

            /* retract a service
             */
            env.retract(key);
        }
        catch(ObjectExistsException oee)
        {
            /* service could not be published */
        }
        catch(NoSuchObjectException nsoe)
        {
            /* service could not be retracted */
        }
    }
}
```

Based on the example, the service class can alternatively be published, looked up, or retracted directly on the shell layer.

```
shell> publish sicari.modules.ServiceImpl /public/my_service
shell> print /public/my_service
shell> retract /public/my_service
```

10 Outlook

The purpose of this documents was to provide the SicAri platform's architectural specification. Starting from a high-level architecture of the platform, the specification has been refined. The objectives of the central components, such as the SicAri kernel, have been described and the

security architecture has been explained. The basic and application services of the SicAri platform have been specified from a use case, logical, and deployment view. Finally, this document gave programming guidelines and examples for the development of the SicAri prototype and its services.

In order to reduce the complexity of this specification, details about the particular service's methods and parameters have been ommitted here. More detailed programming guidelines and the definition of SicAri Java APIs will be provided in a *SicAri Developer's Guide* [4].

References

[1] J. P. Anderson. Computer Security Technology Planning Study, Volume II, ESD-TR-73-51. Technical report, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA, 01730, October 1972.

[2] SicAri Consortium. The Logging Wrapper. Internal Project Report, 2004. CVS Path: `/sicari/prototype/doc/logging/`.

[3] SicAri Consortium. The SicAri Code Conventions. Internal Project Report, 2004. CVS Path: `/sicari/prototype/doc/codestyle/`.

[4] SicAri Consortium. The SicAri Developer's Guide. Internal Project Report, 2004. CVS Path: `/sicari/prototype/doc/develop/`.

[5] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Computer Security Series. Artech House, Boston, 2003.

[6] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001. URL: `http://csrc.nist.gov/rbac/rbacSTD-ACM.pdf`.

[7] Li Gong. *Java^{TM} 2 Platform Security Architecture - Version 1.2*. Sun Microsystems Inc., 2002. URL `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-s% pec.doc.html`.

[8] Peter Gutmann. *Cryptographic Security Architecture – Design and Verification*. Springer, New York, 2004.

[9] Sun Microsystems Inc. Security Code Guidelines. Technical report, Sun Microsystems Inc., February 2000. URL: `http://java.sun.com/security/seccodeguide.html`.

[10] National Institute of Standards and Technology (NIST). Role-Based Access Control. URL: `http://csrc.nist.gov/rbac/`.

[11] Jan Oetting, Taufiq Rochaeli, and Ruben Wolf. Requirements for the SicAri Architecture. Technical report, SicAri Consortium, 2004. URL: `http://www.sicari.de`.

[12] Taufiq Rochaeli. Security Policy Specification of SicAri Scenarios. Technical report, SicAri Consortium, 2004. URL: `http://www.sicari.de`.

[13] Taufiq Rochaeli and Ruben Wolf. Requirements on Sicari Security Policies. Technical report, SicAri Consortium, 2004. URL: `http://www.sicari.de`.

[14] Ravi Sandhu. Role activation hierarchies. In *Proceedings of the third ACM workshop on Role-based access control*. ACM Press, 1998.

[15] Sun Microsystems. *Java 2 Platform Std. Ed. v1.4.2 API documentation, Class java.lang.SecurityManager*, 2004. URL: `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/SecurityManager% .html`.

[16] Sun Microsystems. *Java Tutorial – Providing Your Own Security Manager*, 2004. URL: `http://java.sun.com/docs/books/tutorial/essential/system/security% .html`.

[17] Ruben Wolf and Markus Schneider. Context-Dependent Access Control for Web-Based Collaboration Environments with Role-based Approach. In *Computer Network Security, 2nd Intern. Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2003*, volume 2776 of *LNCS*, pages 267–278, St. Petersburg„ Sept. 2003. Springer.

[18] Ruben Wolf, Markus Schneider, and Thomas Keinz. A Model for Context-dependent Access Control for Web-based Services with Role-based Approach. In *DEXA Intern. Workshop on Network-Based Information Systems, NBIS 2003*, pages 209–214, Prague, Sept. 2003. IEEE Computer Society Press.