UNIVERSITÀ DEGLI STUDI DI CATANIA
Dipartimento di Ingegneria Elettrica, Elettronica ed Informatica

Dottorato di Ricerca in Ingegneria Informatica e delle Telecomunicazioni
XXVI Ciclo

---

# SECURITY AND PRIVACY POLICY MANAGEMENT IN DYNAMIC AND CONTEXT-AWARE USER-CENTRIC SYSTEMS

---

Ing. Salvatore Monteleone

| Coordinatore | Tutor |
|---|---|
| Chiar.ma Prof.ssa V. Carchiolo | Chiar.mo Prof. V. Catania |

*to my family and Lidia*

# SOMMARIO

Prendendo in considerazione i sistemi *user-centrici*, il problema del controllo dell'accesso alle risorse riservate è ancora aperto, soprattutto a causa della grande quantità di dati generati ogni giorno dagli utenti. I dati prodotti, sono principalmente smistati dai dispositivi mobili che rappresentano il pi grande punto di raccolta delle informazioni personali, oltre che fonte dei contenuti generati dagli utenti e strumento di controllo della vita sociale in Rete. Tali dati possono provenire anche da altri dispositivi utilizzati quotidianamente, connessi alla Rete e appartenenti a diversi domini (es. sistemi in-car, televisori). In uno scenario multi-dominio cross-piattaforma come quello presentato, in cui i browser rappresentano il principale veicolo di diffusione delle applicazioni, attualmente manca un sistema coerente per il controllo degli accessi orientato alla sicurezza e alla tutela della privacy. Per far fronte a questo problema è stata sviluppata una piattaforma che mira a fornire un unico sistema di gestione delle policy che sia cross-device ed in grado di funzionare su dispositivi web-enabled, come i moderni televisori, handset, sistemi in-car e PC. Questa piattaforma, denominata *webinos*, risolve le carenze esistenti introducendo il concetto di

*personal zone* che consiste nella virtualizzazione dell'insieme di tutti i dispositivi e servizi di proprietà di un particolare utente. Tutti i dispositivi di una zona possono sincronizzare le loro politiche di controllo degli accessi e possono creare regole flessibili fatte su misura per il singolo utente, il dispositivo o l'intera zona.

In questa dissertazione sono presentati i dettagli architetturali e le conclusioni maturate durante la progettazione della piattaforma in questione.

# ABSTRACT

Considering *user-centric* systems, the problem of controlling the access to reserved resources is still open, especially due to the big amount of personal data generated every day by users. These data come primarily from mobile devices that became the main repository of personal information and source of user-generated contents as well as the principal controller of the social networked life. These data comes also from other connected devices of the everyday life, belonging to different domains (e.g. in-car systems, TVs). In a multi-domain cross-platform scenario, in which web browsers represent the main vehicle for apps, currently there is a lack of consistent access control for security and privacy. To cope with this, a platform which aims to provide a single, cross-device policy system for web applications on a wide range of web-enabled devices including TVs, handsets, in-car systems and PCs has been developed. This platform, named *webinos*, solves the existing deficiencies in web authorisation by introducing the concept of a *personal zone*, the set of all devices and services owned by a particular user. All devices in this zone can synchronize their access control policies through interoperable middleware and can create flexible rules

which may refer to an individual user, device or the entire zone.

Details of the architecture are provided and also the explanation of how the experience during design highlighted several conceptual challenges.

# CONTENTS

CHAPTER

# ONE

# INTRODUCTION

Protecting our personal data from unauthorised access and private data disclosure is a very important task, especially in recent times when the rapid concurrent evolution of mobile computing and social networking expose users to new risks in terms of loss of control of their data.

One of the factors that contributed to the success of some mobile operating systems, such as Android and iOS, is the availability of application markets which provide a very large number of applications. In this dynamic scenario, malicious applications - and the ones which do not take care about users' privacy - may access private data, manipulating and spreading them uncontrollably. These applications may also access system's functionalities impacting, for example, on user's bill without his conscious consent (e.g. using SMS, call and Internet services especially while roaming). Modern mobile operating systems commonly prompt the user with an authorization dialog showing the

list of functionalities which an application will have access to. This prompt appears only during the application's installation process and it is not possible to define access control policies to be enforced at run-time. In addition, the user can only decide to install or not a certain application without any degree of flexibility. It is not possible, for instance, to decide to install an application and then constraint the access to some functionalities during run-time in the case some specified conditions occur.

Problems due to lack of coherent access control or private data disclosure are frequent also in those systems in which applications run inside a web browser, since this kind of applications use Javascript APIs to access native OS features and these APIs are implemented breaking the browser's sandbox security model. This scenario is common to many internet-connected devices, especially those which run cross-domain / cross-platform applications and the solutions working within it can also be exported in user-less domains such as Machine-to-Machine (M2M), Internet of Things (IoT) and Web of Things (WoT).

## 1.1 Structure of this Dissertation

In this work are presented some solutions intended to cope with security and privacy issue as those enumerated above.

This dissertation is organised in seven chapters (including this introduction) as follows:

Chapter 2 provides basic definitions related to access control and privacy policy management. It also presents an extendible framework for policy management named XACML.

Chapter 3 presents *SecureDroid*, an extension of the Android security framework able to enforce flexible and declarative security policies at run-time, providing a fine-grained access control system. In particular, this extension focus on context dependent policies that allow the user to specify the way in which applications work accordingly with context.

Chapter 4 presents a cross-platform policy management system for web applications. This system has been designed and implemented within an open source platform, named *webinos*, which aim is to support and simplify the development of multi-domain cross-platform applications, while preserving users' resources from unauthorised use and disclosure.

Chapter 5 explores the feasibility of using the "browser paradigm" within M2M, proposing *webinos* and it's security model as a solution to provide a secure multi-user and cross-domain approach to the Web of Things (WoT), and also an effective way to address some of the most common issues in such heterogeneous scenarios.

Chapter 6 proposes how the designed solutions have effect also in systems where the users are able not only to generate information but also to generate (and provide) services that can be consumed by other users or social/aggregation applications.

Chapter 7 summarizes the conclusions and proposes further works related to the presented subject.

## 1.2 Acknowledgements

Part of the results described in this dissertation comes from the research funded by the EU FP7 *webinos* project (FP7-ICT-2009-05 Objective 1.2).

The code produced while working on this project is freely available at [1] and has been forked from / contributed to *webinos* project repositories [2, 3]. Requirements, specifications and all the other deliverables are available in the project's site.[1]

[1] http://www.webinos.org

CHAPTER

# TWO

# TECHNICAL BACKGROUND

This Chapter presents some basic definitions that explain the meaning of access control, data handling and policy management in the IT field. It also introduces some fundamental concepts and a standard, namely XACML, which is globally recognised as a reference for any work in this area.

## 2.1 Access Control

The access control is a way to constrain the interaction among a subject (a single user or a group) and an object (one or more specific resources). With a strict definition it consists on checking whether a subject has been granted with enough priviledges to access an object, when he wants to access it.

In a wider definition access control includes:

- **Identification and authentication**, to ensure that a subject is who he claims to be when he accesses a system;
- **Authorization**, to specify what a subject can do;
- **Access approval**, to grant access during operations, by association of users with the resources that they are allowed to access, based on the authorization policy;
- **Accountability**, to log what a subject (or all subjects associated with a user) did.

### 2.1.1 Access control models

Access control models can be classified into two categories: those based on capabilities and those based on access control lists (ACLs). In a capability-based model, the access to an object is granted by the possession of a reference or capability to the same object. Access is conveyed to another party by transmitting such a capability over a secure channel. In an ACL-based model, a subject's access to an object depends on whether its identity is on a list associated with the object. Access, in this case, is conveyed by editing the list (different ACL systems have a variety of different conventions regarding who or what is responsible for editing the list and how it is edited).
Both capability-based and ACL-based models have mechanisms to allow access rights to be granted to all members of a group of subjects (often the group is itself modeled as a subject).
Access control models can also be classified as discretionary or non-discretionary. The three most widely recognized models are Discre-

tionary Access Control (DAC), Mandatory Access Control (MAC), and Role Based Access Control (RBAC). MAC is non-discretionary.

### Attribute-based access control

In attribute-based access control (ABAC), access is granted not based on the rights of the subject associated with a user after authentication, but based on attributes of the user. The user has to prove so-called claims about his attributes to the access control engine. An attribute-based access control policy specifies which claims need to be satisfied, in order to grant access to an object. For instance the claim could be "older than 18". Any user that can prove this claim is granted access. Users can be anonymous when authentication and identification are not strictly required. One does, however, require means for proving claims anonymously. This can for instance be achieved using anonymous credentials. XACML (extensible access control markup language) is a standard for attribute-based access control. XACML 3.0 was standardized in January 2013.

### Discretionary access control

In Discretionary access control (DAC) models, a policy is determined by the owner of an object. The owner decides who is allowed to access the object, and what privileges they have.
In these models every object has an owner. In most DAC systems, each object's initial owner is the subject that caused it to be created. The access policy for an object is determined by its owner. Many operating systems such as those UNIX based use DAC.

### Mandatory access control

Mandatory access control refers to allowing access to a resource if and only if rules exist that allow a given user to access the resource. It is difficult to manage, but its use is usually justified when used to protect highly sensitive information. Examples include certain government and military information. Management is often simplified (over what can be required) if the information can be protected using hierarchical access control, or by implementing sensitivity labels. What makes the method "mandatory" is the use of either rules or sensitivity labels.

### Role-based access control

In a Role-based access control (RBAC) model access policies are determined by the system, not the owner. RBAC is used in commercial applications and also in military systems, where multi-level security requirements may also exist. RBAC differs from DAC in that DAC allows users to control access to their resources, while in RBAC, access is controlled at the system level, outside of the user's control. Although RBAC is non-discretionary, it can be distinguished from MAC primarily in the way permissions are handled. MAC controls read and write permissions based on a user's clearance level and additional labels. RBAC controls collections of permissions that may include complex operations such as an e-commerce transaction, or may be as simple as read or write. A role in RBAC can be viewed as a set of permissions.

## 2.2 Policy Management

Access control and data handling are constrain by the enforcement of one or more policies. Policies are not static objects, they have a lifecycle which is usually controlled by a component named Policy Manager. A Policy Manager handles all the different phases that characterise policies' lifecycle (policy specification, deployment, editing, reasoning and enforcement) and acts to collect all the information needed in that process.

## 2.3 XACML

The eXtensible Access Control Markup Language (XACML) is standard, provided by OASIS, that defines an XML access control policy language and a policy enforcement architecture based on the exchange of request/response messages. This standard also describes the format of all the specified messages and how to evaluate them, but it does not promote a specific implementation strategy since it is intended to support the interoperability between access control solutions provided by different vendors.

The root element defined by XACML is a PolicySet which is made up by both a set of policies (containing at least one policy) and a Policy Combination Algorithms (PCA) which take the authorization decision from each policy as input and apply some standard logic to come up with a final decision. Each XACML policy is in turn made up by a Target and a set of Rules. A Target is basically a set of simplified conditions for the Subject, Resource and Action that must be met for a PolicySet, Policy or Rule to apply to a given request.

Each Rule represents the core logic for a policy and it is made up by a Condition (which is a boolean function and may contains nested sub-conditions) and an Effect (a value of Permit or Deny that is associated with successful evaluation of the Rule).

The Data-flow model introduced by XACML is shown in Figure 2.1. In short, a Policy Enforcement Point (PEP) is responsible to intercept a native request for a resource and to forward this to the Context Handler (CH). The CH will form an XML request based on the requester's attributes, the resource in question, the action, and other information pertaining to the request. The CH will then send this request to a Policy Decision Point (PDP), which will look at the request and some policy that applies to the request, and come up with an answer about whether access should be granted. That answer (in XML format) is returned to the Context Handler, which translates this response to the native response format of the PEP. Finally the PEP, based on the response, allows or denies access to the requester.

An XACML policy can be represented as $p = \{t, R\}$ where $t$ is the policy's Target and $R$ is the set of policy's Rules. Each Target can be represented as the triple $\{s, a, r\}$ where $s$ is the subject, $a$ is the action and $r$ is the resource. Each Rule $r_i \in R$ can be represented as the pair $\{e, c\}$ where $e$ is the effect (it mainly assumes values '+' to indicate Permit or '-' to indicate Deny) and $c$ is the condition.

The complete flow depicted in Figure 2.1 is:

1. The Policy Administrator defines policies and policy sets at the Policy Authorisation Point (PAP).

2. The Service Requester issues a request to the Policy Enforcement Point (PEP) to access the specified resource. This requires

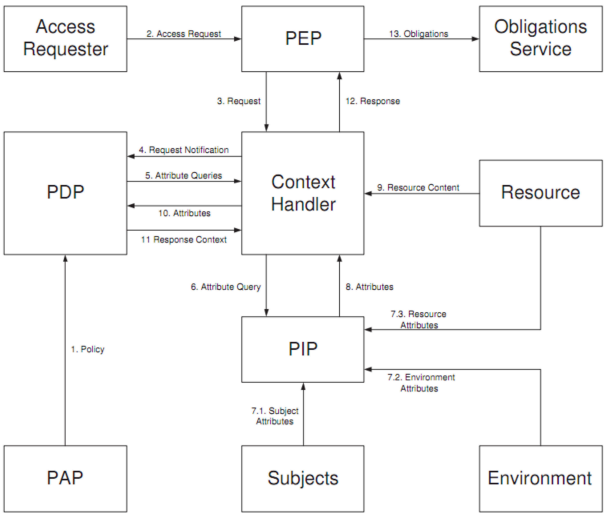


***Figure 2.1:*** *XACML Data-flow model*

fetching the attributes and policies associated with the resource, the action, the environment, and the service requester.

3. The PEP sends the request for access to the XACML Context Handler in native request format. This may include the details of attributes of the subjects, resources, actions, and environment.

4. The Context Handler creates an XACML request context and sends a policy evaluation request to the PDP.

5. The PDP queries the Context Handler for attributes of the subject, resource, action, and environment needed to evaluate the policies.

6. The Context Handler obtains the attributes either from the request context created in step 4, or it queries a PIP for the at-

tributes.

7. The called PIP collects attributes about subject, resource and environment.

8. The PIP returns the requested attributes to the Context Handler.

9. Optionally, the Context Handler includes the resource in the context.

10. The Context Handler returns the requested attributes to the PDP. PDP continues evaluating the policy as attributes are made available.

11. The PDP sends the response context (including the authorization decision) to the Context Handler.

12. The Context Handler responds to the PEP, after translating the response context to the native response format of the PEP.

13. The PEP executes any relevant obligations.

A more detailed description about XACML is provided by [4].

CHAPTER

# THREE

# EXTENDING ANDROID SECURITY MODEL

## 3.1 Scenario

Mobile devices became in last decade the central hub for our personal information due to the amount of private data they might control locally (e.g. address and phone books, texts and emails) as well as the capability to manage remote information and systems (e.g. again emails, social networks, cloud storages and enterprise systems). In addition, smartphones are one of the main sources of personal information that can feed remote systems like social networks. Examples of this information are the photos and videos recorded on the devices and the multitude of data that might be gathered by sensors like GPS, accelerometer, etc. To manage all these data and obviously for other purposes, users make an extensive use of applications: pieces of software that may also substitute operating system modules for some core operations and functionalities (e.g. configuration, keyboard). One of

the factors that contributed to the success of some mobile operating systems like Android and iOS is the availability of application markets, which provide a very large number of applications. Malicious applications - and the ones that do not take care about users' privacy - may access private data, manipulating and spreading them uncontrollably. These applications may also access system's functionalities impacting on user's bill without his conscious consent (e.g. using SMS, call and Internet services, especially while roaming). Modern mobile operating systems commonly prompt the user with an authorization dialog showing the list of functionalities which an application will have access to. This prompt appears only during the application's installation process and it is not possible to define access control policies to be enforced at run-time. In addition, the user can only decide to install or not a certain application without any degree of flexibility. It is not possible, for instance, to decide to install an application and then constraint the access to some functionalities during run-time, in the case some specified conditions occur.

Modern mobile operating systems allow users to install applications in an easy way. This feature opens big chances for users that are now able to customize devices following their needs and moods but on the other side, it poses strong security and privacy concerns. Smartphones other than handling personal and reserved data, provide also services which have a cost for users. The resources that require protection can be distinguished into:

- User's personal/reserved data: address and phone books, call lists, stored messages, passwords, codes, etc.;
- Applications' reserved data: every application has files related

to its configuration or its data, that should be reserved (like passwords). An unwanted access can modify the behavior of the application, its configuration and its stored data;

- System's and device's reserved data: configuration and all the information related to the device (e.g. IMEI, model);
- System functionalities: making calls, sending messages, connecting to Internet, localizing the user are just examples of functionalities that the system provides to applications and that can affect user's privacy or can lead to unwanted expenses. These functionalities have to be provided only to those applications that effectively required them and only following user consensus;

The choice adopted by Android is that each application states, during its installation process, all the system's permissions that it will require at run-time. The user can give his assent to allow the application to use requested functionalities or stop the installation. In other words, Android bases its security permission granting system on a "prompting" approach at installation time. In addition, the user does not have a centralized control (thus offered by the system) to change the behavior of applications while they are running. This type of control can be only provided independently by single applications. The ability to grant or to reject authorizations to an application at run-time goes against one of the principles of usability that Android developers have decided to follow:

> Android has no mechanism for granting permissions dynamically (at run-time) because it complicates the user

experience to the detriment of security[1].

It is easy to understand that exposing a user to repeated grant requests could seriously damage the device usability. However, it is possible to consider a trade-off between interaction complexity and access control granularity so that a policy-based mechanism to specify a priori the behavior of applications should be introduced. In [5], authors examined the efficacy of privacy signaling provided by Android during installation process. As a result of their research they found that the average user does not pay much attention to warning messages, as he often is not able to understand the consequences that may arise allowing a set of permissions to an application. Conversely, the "download count" information related to an application has a strong effect on the users' decisions. We believe that the Android security model may be enriched with new functionalities. First of all, more and more applications offer a lot of functionalities and then require a large number of permissions during their installation. Users may be interested only in a subset of these functionalities, which might require a limited number of permissions. In the case, for example, of a photo editing application, the user may be only interested on storing photos and not on geotagging functions. In this scenario the user is forced to consent to all of the requirements of the application, otherwise the system would prevent the installation. Moreover, latest generation of mobile devices are able to obtain information about user's context. Information from sensors such as GPS, accelerometer, gyroscope can be used together with information coming from social networks or provided by OS, to determine user's context in a specific instant. We believe that all this

[1] http://developer.android.com/reference/android/Manifest.permission.html

user's information can be used by the system to allow the user to decide how his applications should behave according to what his context is. As an example, user might specify the following policies

- Application X does not have the rights to access Internet if the device is roaming
- Service Y must be halted if battery level is lower than 10%

These context-aware policies could be defined by users for each installed application. Another interesting scene could be the one where context-aware policies are not chosen by the user but are imposed to him by other authorities. As an example, we could consider an IT department of an enterprise that may like to impose some policies on employees devices (for instance only during work hours or only to some applications or only when the device is within company site): in this case we need a very flexible policy language and enforcer in order to express more complex and context-aware polices. The needed flexibility becomes even more pushing if we consider the same context, where a Bring-Your-Own-Device [6] policy is in place, and we need to adapt policy based on type of applications, context - private or enterprise - and source of applications. In the consumer segment, other examples can be a policy enforced by a museum to its guests to prevent them using smartphones' camera inside some rooms, or again policies which constraint access to sensors and network communication, etc.. Here, we therefore propose an extension of the Android security framework that is able to:

- apply security policies at run-time (other than during installation-time)

- enforce security policy and user choices taking into account some contextual information.

- use well-known standard languages for the definition of such policies

- allow the user to choose and change the behavior of such policies

We will also present the main research areas, which can address and improve our proposal.

## 3.2 Related Work

The growing of context-aware services has accelerated academic interest toward context-aware policies especially for those scenarios which put mobile devices under the spotlight. Some important contributions in this field have been directed through the definition of policy models suitable for context-aware scenarios [7]. Access control systems should be able to support and understand any new context information in order to address access control requirements. To make this possible, Cheaito et al. presented an extensible access control solution based on XACML making it able to understand new attributes data types including the functions that are used in the policy to evaluate users' requests [8]. Another interesting work is [9] where Li et al. proposed an access control policy model based on context and role that can be appropriate for web services. The model takes context as the center to define and perform access control policies. It uses the contexts of user, environment and resource to execute dynamic roles

assignment and constrain the authorization decision. Another interesting work, which addresses conflict problems in context-aware policies, is [10] where authors propose a framework where authorization for a particular access request is decided dynamically based on context information. They further support dynamic conflict resolution where current policy is chosen at run time based on context information. Finally, the emerging of context aware services and relative mobile applications is making it necessary to redesign actual mobile OS's security frameworks. Current smartphone systems marginally allow users to specify the behavior of the applications through the presence of contextual information, Conti et al. propose CRêPE [11], an Android security extension which allows context-related policies to be set (even at runtime) by both the user and authorized third parties, locally or remotely. Policies which can be defined in CRêPE are based on the status of variables sensed by physical (low level) sensors, like time and location. Apex [12] is an extension of the Android permission framework which allows users to specify detailed runtime constraints to restrict the use of sensitive resources by applications. The user can specify his constraints through a simple interface of the extended Android installer introduced by authors called Poly. Within Android, Google provide a so-called "Device Policy for Android" environment that allows to set policies to enforce use of PIN or password and screen lock on the device and allow an administrator to wipe the device remotely. This framework has the only aim of preventing physical access to information on a device which is not under direct user control. It has no relation with the behavior's control of a certain application. In the field of improvements to the security of standard operating systems, it is well-know the case of Security Enhanced Linux [13] where

on top of Linux an amended security framework was introduced to enhance the capabilities of the operating system. SELinux became part of the standard Linux distribution once adopted by the community. This work goes in the same direction, introducing a more featured policy system on an already available platform.

## 3.3 Access control in mobile OSs

Mobile operating systems protect their data and functionalities using several approaches. Digital signature and certification, for example, guarantee authenticity and integrity for the applications which are being installed. Using a central repository, like an Application Market, besides providing to users the possibility to find every kind of applications, allows them to know information about the application's author and to read other users' comments about the application, which might reveal possible bugs and unexpected or malicious behaviors. Another fundamental method to guarantee applications' security is the isolation: every application is executed in a restricted execution context where it can access only a minimum set of functionalities: it cannot access reserved data and reserved functionalities, it cannot communicate with other applications or access their data. The way in which the main OSs such as iOS, Android and Windows Phone manage the security of their applications is undoubtedly one of the factors that have contributed to their success.

On the iOS platform, each application is signed with an Apple-issued certificate. Applications can be installed only from iTunes market. To publish an application, an author must be registered and

its application is first deeply tested. Every third-party application is sandboxed and can access exclusively to its directory. Accessing to protected data or functionalities and communicating with other applications is possible only using system APIs. An application declares protected functionalities that it needs during market registration phase, so security compliance is established by Apple in registration phase. A user can be advised of which functionalities an application needs during the installation phase. Apple's *App Store* acts as a gatekeeper for the applications uploaded by developers. The Apple's staff checks the source code of uploaded applications and retains the possibility to refuse them if not compliant with the security criteria.

As reported in [2] "The Windows Phone 7 and 8 security model introduced the chamber concept, which is based on the principle of least privilege and uses isolation to achieve it; each chamber provides a security boundary and, through configuration, an isolation boundary within which a process can run. Each chamber is defined and implemented using a policy system. The security policy of a specific chamber defines what operating system capabilities the processes in that chamber can call. Every app on Windows Phone (including Microsoft apps and non-Microsoft apps) runs in its own isolated chamber".

Android is based on Linux Kernel and uses its mechanisms to ensure security. Each application run in a different process as a different user with an UID (User ID), with a different home directory and no root privileges. So application's isolation is done at kernel level and for each application, included system and preinstalled applications. Communication between apps or between apps and system is

[2] "Windows Phone 8 Security Overview" - 2012

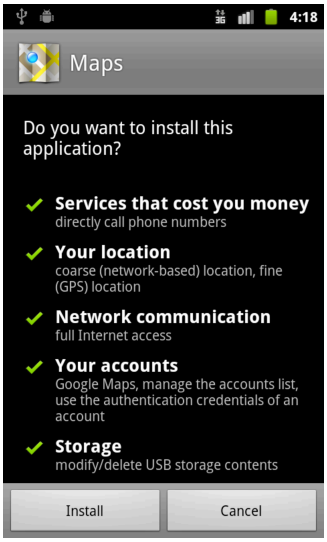


***Figure 3.1:*** *Android installation process*

done with IPC mechanisms that extends Linux kernel, like Binder, or are implemented at higher level, like Services, Intents, and Content Providers. Android has a centralized repository, *Google Play Store*, where application can be published, but can also install applications from other sources. Each application is certified and the certificate is verified during installation. Unlike the App Store, Google Play exercises no control over the source code of applications uploaded by users.

## 3.4 Android Security Framework

Protected data and functionalities are provided by Android to applications using the Android Permission System. Each kind of protected

data or functionality is associated to one permission. Android provides a standard set of permission that includes all functionalities provided by the system. An application can define its own permission to provide and protect its functionalities from other applications. An application lists permissions it needs using a Manifest file, which is read by the system during installation. Android has four protection levels which differ for their relevance: *Normal*, *Dangerous*, *Signature*, *SignatureOrSystem*. Each protection level has to be specified by a developer when he defines own permissions.

During installation, a dialog box listing all the permissions at protection level *Dangerous* requested by the application that the user is trying to install, is shown. User then reads and decides whether to install the application or not. If the application is installed, all permissions it declared in its manifest are registered and associated to the UID that identifies the application itself. Every time an application tries to access to protected functionalities or data, the system controls if the permission is registered for the application's UID. If the answer is positive, the permission is granted, else the application cannot get access. Permission are granted once at install time at cannot be revoked afterwards: the only chance is to uninstall the application. Is not possible to revoke a permission during application's execution. It is possible making a choice only during installation. Moreover all decisions taken are static. There is no connection between policy's decision and user's context: battery level, position, connection, and any other variable are not taken into account during evaluation. Google justifies this static mechanism saying that it would be too boring for users answering to security prompts every time an application carries out an operation, the user would end to ignore advice, granting per-

missions without reading. So Google prefers to concentrate all user attention at install time, where he has much focus on application, its functionalities and problems. In this way, Google takes no account of the correlation between security decisions and context.

Android's security framework checks for permissions when one of the following situations occurs. i) An application want to access to a particular functionality protected by a permission (e.g. GPS information), ii) An application tries to start an activity of another application, iii) Both when an application sends and receives broadcasts, iv) An application tries to access and operate on a content provider and v) When binding to or starting a service.

With our work we try to take into account all of these cases. We have studied the Android source code in order to establish which are the most suitable points to place our run-time policy's check.

## 3.5 Policy Model

We've customized the XACML[3] policy model to suit the Android system. The policies we consider, contain information about subject, resources and context. More precisely, in our model: i) A *Subject* represents the application to which the policy will be applied. ii) A *Resource* represents the Android's permission protected by the policy. iii) *Context* is a set of information which characterize user's or device's context.

Each of the above elements, according to the XACML standard, is identified by a series of attributes.

[3]https://www.oasis-open.org/committees/xacml

A Subject can be specified through several attributes which are taken from application certificate. The most important are:

- *ID*: The name of the application's package,
- *AUTHOR-KEY-CN*: Author's common name,
- *AUTHOR-KEY-FINGERPRINT*: Author's signature.

Resources are expressed with "api-feature" attributes which are mapped to standard Android permissions, and with "uri" in case of accessing a content provider.

Our context model is made up of the following information:

- Location data: indicate user position e.g. latitude and longitude with a radius, or higher level informations like city and nation;
- Device battery data: indicate battery level and if it is charging or not;
- Time data: e.g. time interval or day of week information;
- Connection type used by device at that moment: this information allow to distinguish between connections that costs money to the users and those free of charge;

Our policy model is based on a subset of the XACML grammar for policy definition. Mainly, a policy is composed of a *target* that identifies the entity to which the policy will be applied (in our case the subjects are applications) and one or more *rules*. Each rule contains an *effect*, which can assume values "permit", "deny" or "prompt", and a set of matches for the resources and the environment. An example

of security policy which could be defined inside *SecureDroid* is the following:

The above policy sets the behavior for an application (in this case "exampleApp") denying Internet access toward "http://blockedsite.org*" if the current connection is of type "mobile-roaming". The policy contains "deny-overrides" as rule's combination algorithm and it is composed of two rules which are evaluated in the same order they are written. The first rule contains itself three matches which are evaluated with *and* logic (as default) since the *condition* element does not have a specified value for its *combine* attribute. The first match indicates the resource required by the application (android.permission.INTERNET), the second one specify a filter for the URI ("http://blockedsite.org*") and the last one specifies the device context ("mobile-roaming"). According to deny-overrides algorithm, the PDP will answer "deny" if "exampleApp" requires Internet access while it is roaming, otherwise it will answer "permit".

## 3.6 *SecureDroid*

Controlling the way in which an application works during run-time means changing the normal operation carried out by Android's security framework. A naïve solution to mediate access from applications to Android services would be to extend the Activity redefining its *getSystemService* method. All applications based on this "safe" version of Activity will then be subject to a security check during run-time whenever they try to access a service. Although it may seem a good solution, using this method the control during run-time would take place

```
<policy-set combine="deny-overrides">
  <policy combine="deny-overrides">
    <target>
      <subject>
        <subject-match attr="id"
            match="com.example.exampleApp"/>
      </subject>
    </target>
    <rule effect="deny">
      <condition>
        <resource-match attr="api-feature"
            match="android.permission.INTERNET"/>
        <resource-match attr="uri"
            match="http://blockedsite.org*"/>
        <context-match attr="connection-type"
            match="mobile-roaming"/>
      </condition>
    </rule>
    <rule effect="permit">
  </policy>
</policy-set>
```

only for those applications which make use of this extended version of Activity. All the other applications, such as those in the Android market, would execute without any security control. For these reasons, our choice is to extend the framework based on Android by changing the components which are responsible for granting permissions to applications.

The security engine, here presented, has been developed taking into account one main principle: making access management policies dynamically dependent on the context. The decision process is carried out at each attempt to access a protected resource and its result is derived from the evaluation of a set of policies. The flexibility and modularity of the evaluation method is strongly dependent on these policies, their structure and content. The rules which compose our policies contain a set of attributes dependent on applications and available resources but also some context-dependent. In facts, the huge availability of context information provided by modern mobile devices makes it possible to effectively assess the context in which a user and his device are immersed, allowing to define policies that take into account the variables that characterize surrounding environment and user activity.

By setting up a context-aware security system, it is possible to automate the decision process to reduce the amount of needed user/device interactions, making the whole process transparent to the user (that in this case would be simply notified about decisions taken by the security engine). The definition of context-dependent policies therefore allows to exceed limitations, like those supposed for example by Android's designers, to the usability of a system that continuously prompts a user for resources' access authorizations. It also allows

moving away from the equally limited mechanism of security decisions taken only during an application's installation process. To maintain a high dynamics of the security module proposed, it is also important providing the capability to update during run-time, in a fast and easy way, the access control policies. In addition, we must pay particular attention to operations performed on policy files like editing and management in general. In order to prevent unauthorized, accidental or malicious access to the policies, they are protected, as well as any other resource in the operating system, by the Android's permissions mechanism. In this way, maintaining coherence with system approaches (the basic principle taken into account during the design and implementation of the whole module) it is possible to get at the same time the protection level offered by the operating system to other private resources and the ease of access for authorized entities. It is important to point out that the security engine is an extension module for the Android security framework and it preserves the underlying features and capabilities. Our security engine might grant the requested authorizations to an application after the standard security control made by the operating system is performed, in order to avoid a privilege escalation. An application cannot access to a secured feature without having first declared in its manifest file.

### 3.6.1 Policy Evaluation Order

Despite the fact a smartphone belongs to a user, there are several parties that can control its operations, both in a static and dynamic way. Manufacturers, in either the case that they are owners of the operating system (e.g. Apple) or simply customize it to suit their own devices

(e.g. Samsung), could set policies which restrict access to some functionalities of their devices. A first version of iOS, for example, limited the use of bluetooth to connect only earphones avoiding to exchange of files. Google retains the ability of modifying some aspects of Android devices over the air. Android platform allows not only remote application's uninstallation (via the *REMOVE_ASSET* intent), but also the installation of new applications (via the *INSTALL_ASSET* intent). In addition, Google can push a *REMOVE_ASSET* message down to all the Android phones in order to remote kill a particular application deemed malicious. Also the operator may specify policies on a terminal, which restrict access to certain features or do not allow the use of a device with a SIM card of another operator. The same applies to a third-party, which may require the installation of its own policy on a device in order to limit some functionalities (as in the case of the museum in Section 1). For these reasons, we have provided the opportunity of evaluating multiple policies during the enforcement phase. The policies' evaluation order is quite important because it reflects the priority given by the system to each policy. *SecureDroid* considers the following kinds of policies:

*Manufacturer* $\longrightarrow$ *Operator* $\longrightarrow$ *Third-parties* $\longrightarrow$ *User*

The First to be evaluated is the Manufacturer's policy because it is the most relevant to the system. The last policy to be evaluated is the one that the user can specify for his applications. It should be pointed out that the permissions to add, remove or edit a certain policy have to be granted by an upper level policy. For example if the museum requires adding a policy to the user device, this operation must be allowed at least by the Manufacturer's or the Operator's policies. *SecureDroid* may be adopted also for Bring Your Own Device scenarios,

where companies permit employees to bring personally owned mobile devices to their workplace, and use those devices to access privileged company information and applications. In fact, companies may install on these devices a third-parties policy to avoid unnecessary costs or an incorrect use of devices that can be made by employees. All the policies handled by *SecureDroid* are stored in a system folder, which can not be accessed by normal applications. The only way to modify or add a new policy is through a system service we have developed which is described in Section **??**.

The mechanism introduced by *SecureDroid* starts working only after the standard security check provided by Android. If Android notices that an application is attempting to access a resource for which permission has not been declared in the Manifest, it denies access directly without going through *SecureDroid*. The control carried out by *SecureDroid* is more complex than that performed by relying exclusively on the Android permissions defined in the Manifest. In the worse case, for each request *SecureDroid* has to check $N$ policies, where $N$ is the number of parties which may have control of the device: from the manufacturer to the user. Each of these policies may be more or less complicated on the basis of the number of rules it contains. For all these reasons, placing *SecureDroid* after the standard Android control avoids unnecessary overheads due to all the applications that would be eventually stopped by the system, for example for the absence of a permission in the Manifest file.

### 3.6.2 *SecureDroid Architecture*

*SecureDroid* acts at framework level in the Android system, its basic architecture is depicted in Figure 5.2.

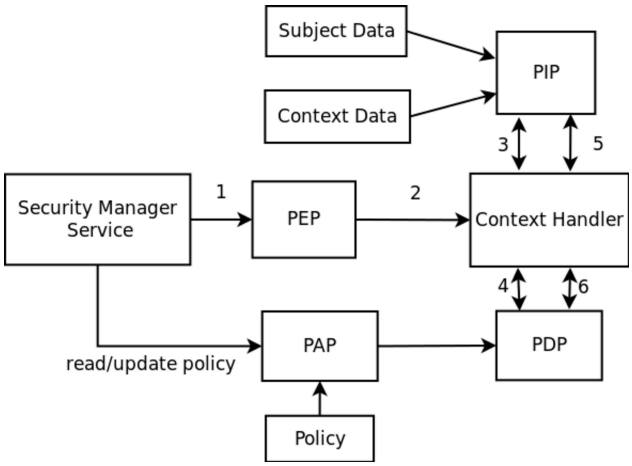


***Figure 3.2:*** SecureDroid *Architecture*

- *Security Manager Service* is the system entry point for our security engine. It's a system service that offers three functionalities: read policies, write policies and make requests. Security Manager maps XACML responses into pre-defined system's responses.

- *Policy Enforcement Point* (PEP) is the enforcement point of policy access control. PEP is responsible for collecting both information about UID of the application which requested access and the permission that is being requested. Finally PEP creates and sends a request to Context Handler.

- *Context Handler* (CH) links all the components, handles both requests from PEP and informations from PIP, finally forwards requests to PDP;

- *Policy Information Point* (PIP) collects informations about subject, resource and context attributes using system Package Manager and device's sensors;

- *Policy Decision Point* (PDP) is the point that reads and evaluates policies. It has principally two functions: given a subject, it reads policy and returns attributes that are necessary for the request's evaluation and, given a complete request, it evaluates that and and the policy returning a decision;

- *Policy Administration Point* (PAP) is responsible for managing the policies. It securely store policies and give them to PDP. It is the only one that can have access to policies, so it is invoked by Security Manager to modify, to read and to store policies.

Security Manager deals with creating and maintaining a reference for all the parts making up the engine. Particular attention is required during the creation of the PAP, which controls if there is a previous policy stored in the filesystem and, if so, loads it. In case of absence or corruption of a policy file, PAP uploads a default policy and passes it to the PDP that will be able to use this policy to perform its functions. When an application requests a resource, such as a system capability or some information from a content provider, its request is checked at run-time by both ActivityManager and PackageManager which are defined in the Android security framework. After the PackageManager

has run the standard checks on permissions declared in the application's Manifest, it starts running *SecureDroid* calling the *doRequest* method provided by the Security Manager. The request which arrives to Security Manager Service contains three informations: 1) the UID of the application which is requesting access, 2) the permission the application is asking for and, 3) an optional URI indicating the resource that is being requested (e.g. a contact, a picture).

Since Android checks only if some permissions have been previously granted to an application, it doesn't provide a way to propagate, trough ActivityManager and PackageManager, information about requested resources (URIs). We have extended this mechanism carrying URIs information up to the Security Manager. In this way, *SecureDroid* can provide a fine-grained control not only based on the permissions declared in the Manifest but also based on the resource itself.

Assuming that user has defined a policy (for example the one in Section 3.5) for a certain application, the execution flow of *SecureDroid* is shown in Figure 5.2. When the application requires access to a resource (for example, when it attempts to open a connection to a URL) the PackageManager checks if the permission *android.permission.INTERNET* has been granted to the application by the user, during installation. If not, PackageManager immediately denies the access, otherwise it invokes the method *doRequest* provided by the Security Manager which calls the PEP. The PEP collects information about the request (the UID of the application, the resource to which the application wants to access and, where the requested URI) and sends them to the Context Handler. This sends the UID to the PIP which returns additional information about the subject (i.e. the application) such as package name, author's signature, etc. At this

stage, CH sends these subject information to the PDP which, knowing the policies, returns to CH a list of context information needed by the PDP to make a decision (e.g. "Is the device roaming?"). CH then requires to the PIP the current values for these context information. The PIP returns the current context status (e.g. "Current connection type is roaming"). At this point, CH owns all the information it needs (subject, resource and context). It sends this information to the PDP which evaluates the policy and takes a decision which is propagated up to the PEP, which eventually enforces it.

### 3.6.3 Decision handling

The PDP takes a decision only once a request from an application arrives, this decision is then propagated to the Security Manager that is responsible for enforcing it. According to policy specification, a decision could be one of: PERMIT, DENY, PROMPT or UNDEFINED.

In the case of PERMIT, Security Manager communicates the positive outcome to the Package Manager, which continues its normal execution, including assessment of the request in the standard mode provided by the system and returns its standard answer.

In the case of DENY, the Package Manager returns a negative response to the application which asked for a permission. *SecureDroid* returns a value different from that returned by the standard control system when a request from an application is not granted. In this way, the Security Manager Service can differentiate whether a request has been denied by the Android standard security framework or by *SecureDroid*.

In the case of PROMPT, a dialog is displayed asking the user

to grant or not a permission to the application. According to policy definition, *SecureDroid* handles three different kinds of prompt: i) *PROMPT-ONESHOT*: the dialog appears every time the application tries to access a function protected by the policy, ii) *PROMPT-SESSION*: the choice made by the user is saved until the system reboot or the device is turned off. iii) *PROMPT-BLANKET*: the choice made by the user is stored in the filesystem and remains in effect until the user decides to remove it. Figure 3.3 shows some screenshots of the dialogs which the system launches when the decision taken by PDP is PROMPT.

*SecureDroid* defines a new exception called *PolicyDenyException* which is an extension of standard *SecurityException* provided by the system. Applications which are aware of the presence of *SecureDroid* are able to manage instances of denied permission. These applications can, in fact, catch PolicyDenyException and understand when *SecureDroid* denies a permission. In this way applications can modify their behavior according to which permission has been denied. Other applications, which do not handle PolicyDenyException, in the case of having access to resources denied by *SecureDroid*, are treated by the system as those applications that try to access a system's capability without having the correct permission in the Manifest. As we have seen, Android grants permissions during the installation phase. So once permission has been given in the Manifest file, an application will always have access to the permissions declared. So it is unusual for applications to see a permission denied during its execution. There is no common way to manage these situations and there are not guidelines from Google. It is not possible to predict *a priori* the behavior of an application when a permission is denied: probably if the application

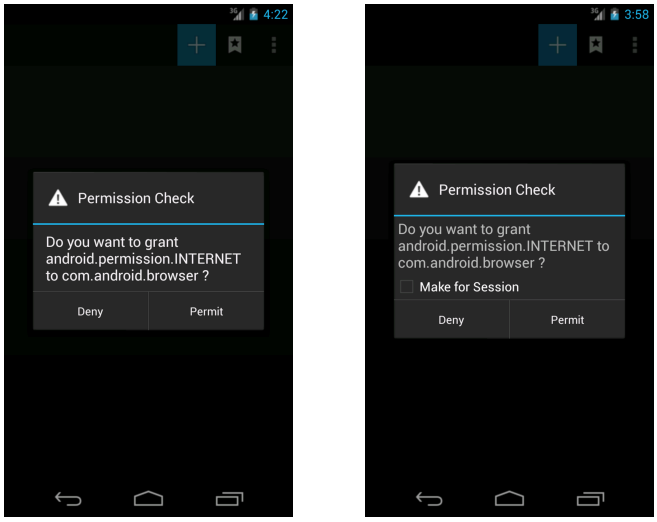


***Figure 3.3:*** SecureDroid *dialogs in the cases of PROMPT-ONESHOT and PROMPT-SESSION*

does not handle this occurrence it will crash.

### 3.6.4 Comparison with other security frameworks

Studying other modern solutions to this problem, we found two approaches which attempt to solve the same problems that were mentioned. In particular, we have seen that the works that most resemble *SecureDroid* are Apex and CRêPE.

Apex is an extension of the Android security framework which allows the user to have more control during application installation. At this stage, in fact, the user can specify security options for each permission required by the application being installed. Apex defines a new syntax for the policy, we have preferred to rely on an already well-

established standard such as XACML, also used by another ongoing project called *webinos* [14]. Apex also allows users to specify the policy with static attributes (for example, to restrict the use of a resource to a maximum number of times per day, or to deny access to the resource after a certain time of day). From our point of view it is important allowing the user to specify policies that depend on his context.

CRêPE has more than one point in common with *SecureDroid*. Like our solution, CRêPE offers the ability of specifying parameters related to user's context while defining policy. Also CRêPE acts at the system level but, unlike *SecureDroid*, performs its checks before the standard permission control offered by Android. We preferred to place our control after that standard to avoid repeated checks on requests made by the system itself. These kinds of requests should be always accepted by *SecureDroid* in order to not destabilize the whole system. In addition, it might be pointless to grant or not a particular permission at *SecureDroid* level (thus evaluating the entire chain of policy that we discussed) if we eventually find out that the application did not declare such a permission inside its Manifest file. Another difference between our framework and CRêPE is that the latter takes into account both access control policies and obligation policies. An obligation policy defines an action that the system will perform under certain conditions. For example it could be possible to automatically disable certain features of the system (such as the camera) when the battery charge drops below a certain threshold. The obligation policy, in the case of CRêPE, can be activated when certain conditions, which depend on the user's context, are met. To support obligation policies CRêPE must perform an onerous procedure. Initially CRêPE gets information about those parameters that characterize context and

thresholds that would trigger the action. Subsequently CRêPE starts monitoring these parameters to detect, moment to moment, if one of the conditions related to the obligation policy is met and, if that happens, it triggers the action. This continuous monitoring activity could be computationally heavy and can effect the energy consumption of the device. Keeping always on the accelerometer, GPS or WiFi etc. can drastically reduce the battery life. Unlike CRêPE, *SecureDroid* does not perform a repetitive control of context variables. This check is performed only at the instant when an application requests access to a resource. In this way we simplify the architecture of the system and reduce energy consumption. In addition to these considerations on energy saving, one of the reasons why we decided to do not take into account obligation policies is because we believe that they should be handled at application level and not at system level.

CHAPTER

# FOUR

# CROSS-PLATFORM POLICY MANAGEMENT

## 4.1 Scenario

In this Chapter is presented a cross-platform policy management system for web applications. The choice fell on web applications because they are gaining access to an increasing amount of important functionalities. Thanks to growing support for new HTML5 standards, such as Javascript APIs for geolocation data, sensors and local storage, web applications have the potential to replace native applications in many situations [15]. Proponents of web applications will also claim that there is a potential security advantage: the web browser provides a sandbox, isolating each page from the local device. This compares favourably with relatively unrestricted native code.

However, the web browser (or web runtime) was not originally designed for complex applications, and its sandbox security model must be broken to accommodate the new Javascript APIs. Unfortunately,

HTML5 API specifications [16] do not provide detail about how authorisation should be implemented, beyond a requirement for consent and revocation, leaving this as an implementation option for each browser or user agent. Considering the number of new APIs under development – access to cameras, address books, calendars, and network information [17] – as well as the wide range of security and privacy implications they have for both individual and corporations, there is a pressing need for new standards for web application access control.

Any access control system for web applications must take into consideration the fact that people own and use several different devices, often at the same time. More types of device are becoming web-enabled, including smartphones, tablets, in-car systems (such as the BMW ConnectedDrive [18]), games consoles and smart TVs. Each will need to implement authorisation for new HTML5 API requests, taking into account the unique user interface and interaction patterns that the device offers. For example, an in-car system might be voice activated, and notify the user of a new access control request through audio. A television might be more context-sensitive, changing interaction method depending on whether the user was watching a movie or casually channel-hopping. Moreover, the rise of 'second screen' companion applications – which allow TV-watchers to interact with the programme through their smartphones and tablets – might enable more interesting authorisation decisions affecting both devices. It also seems reasonable that any access control decision made by a user on one device might also be synchronised with others. For example, if a user grants a web application permission to access the address book on their smartphone, they probably want to allow the same application access to the address book on the user's PC. Web application access

control must therefore be flexible enough to support different devices and interactions, while allowing the synchronisation of user policies between each device.

Browsers are beginning to support synchronization of access control decisions (Google Chrome, for example), but these fail to account for the many different contexts of use associated with each device. For example, a 'prompt every time' policy for geolocation data may be appropriate on a smartphone – where current location changes – but would be irritating on a TV. There are also many ways of notifying mobile users about privacy events [19] and the most appropriate method will be context sensitive. These requirements are difficult to address with ad-hoc browser settings, which only allow limited authorisation decisions and synchronisation options.

We assert that users require, but currently lack, a common way of managing with web application permissions. Standard, cross-device access control policies, which can address web APIs while also being transferable to different devices, are needed. This would enable devices to create custom user interfaces but still produce interoperable authorisation decisions, which could then be synchronised between devices. This, in turn, would let users make access control decisions infrequently (where the same decision applies to multiple devices) but would also allow decisions based on device-specific interactions and context.

Here we describe the architecture and implementation of *webinos*: a cross-device web application platform. *webinos* is designed to give web developers access to a standard set of APIs for device features, as well as providing a common policy model for access control. It also aims to give users a 'seamless' experience by providing better con-

nectivity between devices. The platform is targeted at four domains: PC, mobile, in-car and smart TVs. We have developed an XACML-based system which provides a single policy enforcement mechanism for a user's *personal zone* – the devices they own and use. Our main contributions are: the introduction of a personal zone architecture, the definition of a modified XACML policy system to support this abstraction on all devices, and several observations from design experience.

We begin in Section 4.2 by discussing the state of the art in access control and policy enforcement for mobile web applications and web widgets. In Section 4.3 we describe the requirements for a personal, cross-device authorisation system and give an introduction to the *webinos* personal zone model. Following this, Section 4.4 describes the *webinos* policy framework and section 4.5 covers four of the conceptual challenges faced during design and development.

## 4.2 Background

### 4.2.1 Web applications, widgets and browser security

A web application in a mobile context is a web page or collection of web pages delivered over HTTP which uses server or client-side processing to produce an application-like experience within a web browser [20]. In comparison, web widgets are interactive applications for displaying and/or updating local or remote data, packaged to facilitate downloads to a workstation or mobile device [21]. As such, widgets are similar to web applications but are packaged for installation.

Web browsers support a sandbox model of security, isolating each web application from the device. When a web application wants to break this sandbox, it requires either an implicit user consent based on an action (e.g. file uploads through clicking on a button and selecting a file) or explicit consent through some form of prompt (e.g. the geolocation API). This decision may be remembered by the browser for subsequent requests. On most browsers, saved explicit consent decisions can be revoked through a settings page. Web applications are identified by *origin* – their DNS name, port and protocol – and Javascript on any one origin cannot (in general) access any other. Consent decisions, however, are often stored based on hostname [22].

Widgets are run in a *web runtime* which is similar to a browser. Widgets are packaged as zip files, containing (among other things) a configuration file listing the APIs they request access to. These can be used as permission requests in a similar manner to Android application manifests. Widgets can be identified by namespace and id attributes declared in the configuration file, and widget packages can be signed by their author. Widgets are not restricted by the same origin policy, but may use the Widget Access Request Policy instead [23].

### 4.2.2 Mobile applications and access control

The most popular native mobile application systems are Google Android and iOS. Android takes an 'all-or-nothing' approach to authorisation; this involves developers specifying a mandatory access control policy for APIs in the application manifest [24]. At install time, users either agree to grant all requested permissions or none at all. On iOS, applications are granted access to the full system by default, although

users are able to enable or disable access to geolocation data.

The Android policy system has been criticised for being ineffective, although recent results suggest that application permissions are a valuable security mechanism [25]. Several attempts have been made to modify how policies are specified and used. A common complaint is that Android permissions cannot be individually allowed or denied, a feature provided by MockDroid [26]. Apex [27] does the same, as well as supporting controls such as limiting the number of times a permission may be used. The CRêPE system [28] allows the specification of fine-grained context-dependent policies, which may be based on time, location and the originator of the policy. CRêPE also supports optional notification that a particular context has been activated. Reddy et al. [29] make the observation that Android policies are *resource-centric* rather than *application-centric*, and that security would be better served if applications could request access just to common functionality such as 'scan barcode' rather than the whole camera. This is analogous to program-language security features, where only certain methods are made public to external callers. Finally, SEAndroid has been proposed to better sandbox applications, confine privileged daemon processes and provide a central, analysable system policy [30].

### 4.2.3 Bridging the gap: web application security frameworks

Web applications and widgets require standardised APIs for accessing device features, and these are only slowly being implemented by browsers. As a result, several initiatives have been started to speed up standardisation and create cross-platform web application runtime

environments. These have also included security frameworks.

The BONDI security framework for web applications was created by OMTP, a consortium of operators and handset manufacturers. OMTP was eventually enlarged and renamed as the Wholesale Application Community (WAC). The WAC 2.0 framework [31] combined BONDI with work by the W3C Device APIs and Policy Working Group [17]. BONDI uses XACML (eXtensible Access Control Markup Language) to specify widget access control policies. All widgets requesting access to WAC APIs are mediated by the XACML policy enforcement system. XACML is a general-purpose access control policy language based on subjects, resources, actions and conditions [32]. XACML also defines a reference architecture for policy control and enforcement as well as a message schema. To better situate XACML for a mobile environment, BONDI reduced the number of allowed expressions and specified an appropriate dictionary.

A problem with the WAC model is the role of the handset user as a policy authority. Once an application has been granted access to a certain device capability then it can use, transmit, and share it without any further control or restriction. Work by the EU FP7 PrimeLife project [33] addressed this issue by developing tools to protect user data and manage privacy requirements; these tools included a policy language for privacy and data protection based on XACML [34].

Finally, the Chrome browser also supports up-front permission requests by installable web applications (equivalent to widgets) downloaded from the Chrome Web Store. Permissions are split into high, medium and low-risk categories, with APIs for accessing geolocation and browser history marked as 'low'. Geolocation may also be requested at runtime in the same manner as normal web pages. The

Chrome browser supports synchronisation of settings between different devices when the user has logged in with their Google user account. However, the privacy and security implications of synchronising access control decisions and other data with a central provider such as Google are considerable. Users may need assurance of the integrity and confidentiality of their cloud-based data [35], and may wish to chose a synchronising host that they trust. This implies a need for standard cloud data policies to allow users to migrate between different providers.

### 4.2.4 Related literature

Mobile access control policies have been discussed extensively in academic literature, although rarely in the context of web applications. In ubiquitous computing research, Kim et al. [36] have proposed an extended role-based access control model which can take into consideration changes in user context. Roles are useful abstractions in working environments, but are perhaps less appropriate for personal devices and ad-hoc collaborations. Corradi et al. [37] also introduce a context-based access control system and context model. Again, however, the focus is on pre-defined policies for certain contexts rather than user-defined policies for personal resources. Indeed, user-defined policies for ad-hoc interactions are arguably a more complicated problem; Mazurek et al. show that in home environments people need fine-grained access control and want to be able to express policies requiring others to ask before access to a resource is granted [38]. Baldauf et al. give a further survey of context-aware systems [39].

Several authors have suggested modifications to web browsers and their security model, most of which aim to mitigate attacks by ma-

licious web pages. The Gazelle Web Browser [40] aims to protect web applications (defined by origin) from each other by separating them and their resources into different OS-enforced protection domains. This protection also applies to browser plugins. The Tahoma Web Browsing System [41] isolates websites by compartmentalising them into virtual machines. It makes web applications first-class objects, requiring them to be installed and approved before first use. Tahoma mediates all network interaction, and requires web applications to have a widget-like manifest which define any plugins that are required by the application. Finally, Singh et al. [42] introduce the notion of a *user principle* and suggest that all access to certain APIs and actions (geolocation, browsing history) should be considered *owned* by the user and access to them should therefore also be mediated by users.

In native mobile applications, the many improvements have been suggested to the Android permission system, as discussed in section 4.2.2. In addition, Ai et al. [43] propose an alternative synchronisation system for storing and recovering user data via an online service. If applied to permissions, this might be a way to implement cross-device policies. However, their focus is primarily on preventing the loss of user data, rather than multi-device use cases.

### 4.2.5 Summary

Users are required to make access control decisions for both native and web applications. The general principles are the same in both cases – users must mediate access to potentially sensitive resources – although the way in which permission is granted differs. However,

in both cases users end up making decisions out of context, either at install time or first use, and only for one device at a time. Existing systems which are context-aware tend to be aimed at users with one device as part of larger organisations, and systems such as Chrome allow synchronisation but not context adaptation. Yet context is even more important when policies are synchronised: one's security and privacy preferences when installing a mobile application at home may not be same as those associated with using it outdoors. Unfortunately, designing access control policies for different device and contexts is easier said than done, and likely to become more complex for web applications as they become more functionally capable and used on a wider range of devices. Furthermore, additional scenarios involving multiple devices being used together at the same time, and multiple users interacting, increase the need for a common way of expressing access control policies.

## 4.3 Cross-device authorisation

The state of the art described in the previous section mostly focused on one user with one mobile device. This does not reflect how people use technology today: they may rely on several devices in different contexts. This section describes the requirements for access control within a personal network of devices and explains how the *webinos* architecture begins to satisfy them.

### 4.3.1 Requirements for personal cross-device access control

Each user device can potentially run several web applications, each requiring access to local hardware capabilities as well as functionality from other devices. In combination with the current state of the art described in the previous section, we have elicited the following requirements.

1. Users and other stakeholders shall be able to control access by web applications to Javascript APIs. These APIs may allow access to local and remote resources.

2. Users shall be able to create both device-specific and device-agnostic policies.

3. The platform shall provide synchronization for access control policies, so that policies can be described on one platform and enforced on all.

4. The platform shall allow *context-sensitive* access control decisions: e.g. these may change depending on the environment.

5. The platform shall protect user privacy: access requestors shall be able to qualify how they will use the data they are requesting, and users shall be able to express constraints about data disclosure.

Existing browsers and systems such as WAC provide some support for requirement 1, and the Chrome browser supports limited synchronisation to satisfy requirement 3. Context-aware middleware provides

support for requirement 4. The XACML language may be well suited to expressing policies to satisfy requirement 2, and work by Ardagna et al. [34] can adapt this to satisfy requirement 5. However, the combination of these requirements and the cross-device nature of web browsing provide motivation for the development of the *webinos* platform.

### 4.3.2 The *webinos* platform

*webinos* is a cross-device application platform for mobile web applications and widgets. It provides applications with a set of APIs for accessing local resources, such as sensors and contacts, as well as APIs for communication with other devices and services. The platform aims to create a seamless multi-device user experience through data synchronisation and a consistent access control system. *webinos* is supported on four main device domains: PCs, smartphones, in-car systems and set-top boxes. It was primarily aimed at people who use and own many web-enabled devices, with an emphasis on personal and social computing. More detailed personas of anticipated users are available in project deliverables [44], as are detailed use cases and scenarios [45].

The platform was designed with the following high-level goals in mind:

- **Interoperability** of applications across the aforementioned four device domains. Each application can communicate with others on the same device, with another device belonging to the same user, or with an unknown device elsewhere.
- **Compatibility** – achieved through standard Javascript APIs. This allows applications to run on multiple devices with minimal

modification.

- **Security and privacy** for users and application developers.
- **Adaptability** – allowing applications and devices to take advantage of information about the current environment.
- **Usability** – through the creation of a *seamless experience* for users of applications across multiple devices.

### 4.3.3 *webinos* 'personal zones' of devices

The *webinos* platform aims to meet requirements 2-5 from Section 4.3.1 by introducing the concept of a *personal zone*, the set of all devices owned by an end user. It is an abstraction which provides a basis for managing devices, together with the services running on them. From the web application perspective, all devices in the zone support and expose a set of standard APIs for accessing services such as device features (cameras, geolocation), networking with other devices, and cloud services.

Personal zones contain three key components: the web runtime, proxy and hub. The *Web Runtime* is where web applications are executed. The runtime has been extended to provide a set of Javascript APIs which allow applications to join a personal zone, communicate with other devices and access features. These API requests are forwarded to the personal zone proxy.

The *Personal Zone Proxy (PZP)* implements the APIs and provides the communication layer with other devices. It communicates with the hub through a mutually authenticated TLS session and communicates peer-to-peer with other proxies where internet connectivity

is unavailable. Each proxy has its own policy enforcement and decision point.

The *Personal Zone Hub (PZH)* runs on a web server, which may be hosted by a third party or be part of a user's home networking equipment. It is the central point of access to the zone, so that each device can contact others and request access to resources and services. The hub is a repository for synchronized data and policies, and provides policy enforcement for incoming requests. The hub enrols each PZP into the zone by issuing it with a certificate.

The relationship between these components can be seen in Figure 4.1.

As an example, suppose that Alice is using her smartphone and wants to access a media file on her PC. She loads her media player web application, which requests access to the file via a Javascript API provided by the *webinos* web runtime extension on her smartphone. The runtime extension passes the request to the PZP, which passes the request to the web-based PZH. The hub then forwards this to the PZP on Alice's PC, which contains the implementation of the File API. The file is then transferred through the hub back to the media player application on the smartphone.

## 4.4 The *webinos* policy framework

### 4.4.1 Basic architecture

The *webinos* platform provides privacy protection and access control features to meet the privacy and security requirements of web applications and users. This means protecting against malware and other

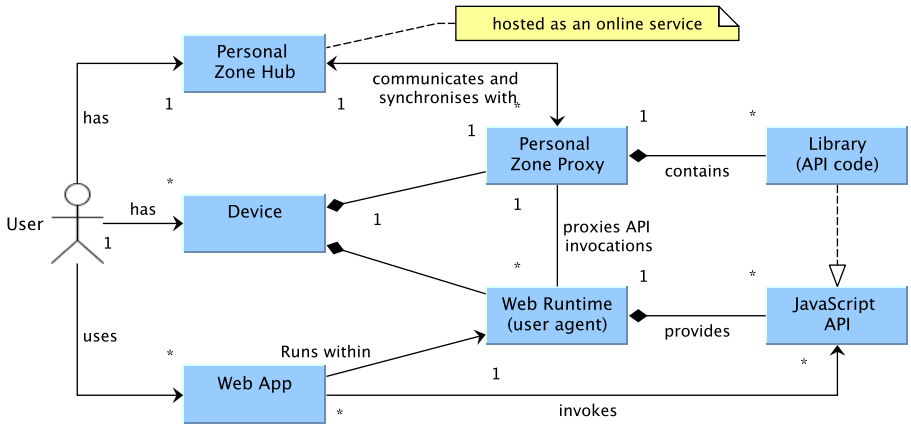


***Figure 4.1:*** *Overview of the* webinos *personal zone model.*

users who may attempt to access more data than they should or simply avoiding the unnecessary disclosure of personal data. To satisfy these requirements *webinos* relies on the strong identification of web applications, users and devices, combined with a least-privilege access control based on XACML.

However, while the latest version of XACML introduced a privacy profile specifying two *purpose* attributes [46], it is still not well suited for describing data handling policies. For this reason the XACML architecture has been adapted using PrimeLife extensions. [47], allowing *webinos* to make access control decisions based on both the request context and user preferences.

The object model depicted in Figure 4.2 shows the main components of the *webinos* policy framework. Both the PZP and PZH enforce policies using standard XACML components, supplemented by:

- The *Decision Wrapper* creates the initial policy enforcement query based on incoming requests.
- The *Access Manager* makes the final decision by combining XACML access control and DHDF data.
- The *Data Handling Decision Function* (DHDF) engine provides privacy and data handling functionalities.
- The *Request Context* manages all contextual information; it stores all the data and credentials released by a user in a given session.
- The *PDP Cache* (PDPC) stores PDP decisions that could be share among personal devices.

### 4.4.2 Adapting the state of the art

Policies are expressed in a similar way to the format defined in BONDI [48]. However, BONDI policies do not support cross-device interaction. The *webinos* framework therefore modifies the *subject* of policies to refer to the device and user. Policies are also able to refer to a dynamically changing set of features, as new APIs may be added by new applications. *webinos* policies are usually composed in the following way, where device T is the *target device* and device R is the *requesting device*:

*User U can access Feature F of Device T through application A on Device R*

Figures 4.3 and 4.4 show an application requesting access to a remote device feature. Every subject (user, application and device)

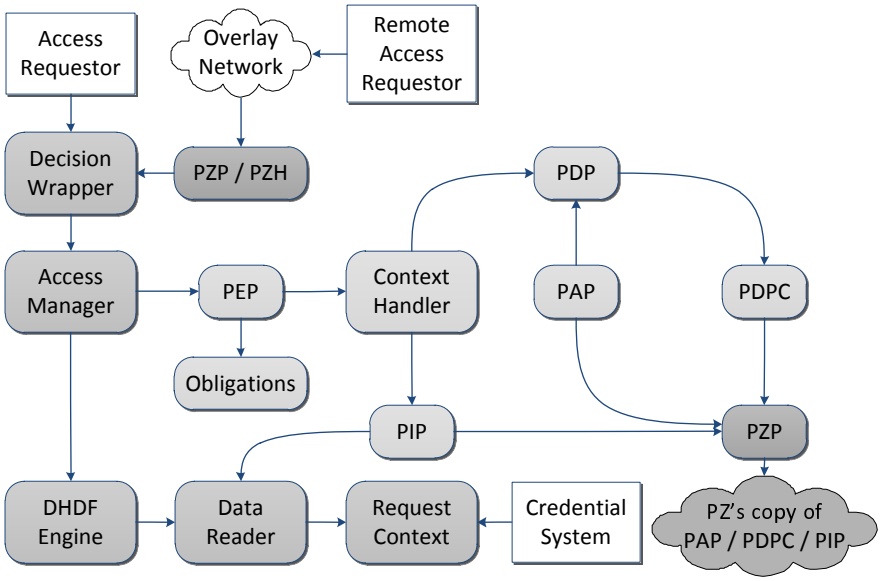


***Figure 4.2:*** *Components of the* webinos *policy framework on a device. Requests come from either local or remote sources, and are then managed by the Decision Wrapper. Policies are held on a PZP and may depend on data fetched from the rest of the personal zone*

has an id used to determine if a specific policy should be taken in account when enforcing incoming requests. To perform this selection the Policy Manager tries to match the request's information against policy targets.

Special URIs are used as ids in policies to match subjects and services. Whenever an entity (user, device, application or service) is registered for the first time, a record with some basic identity information is stored. These records are used to convert a generic ID into a convenient URI.

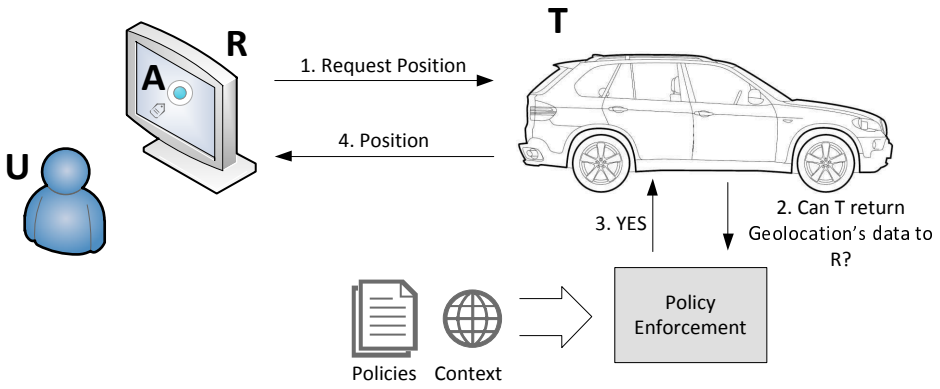


***Figure 4.3:*** *Example of a request to access a remote device's features. Application A launched by user U in set-top box R requires access to geolocation data provided by car T.*

There are four basic types of information in the policy's subject. The *user*, identified by the URI of their PZH. The *requesting device*, as identified by the TLS certificate issued to each device when it enrols in a personal zone; omitting this in a policy allows it to apply to applications running on any device. The *target device*, similarly identified; omitting this allows for policies which refer to an API on all devices, e.g. denying any access to location data. Finally, the *application* ID, author and distributors. This is necessary in device domains (such in-car systems) where only applications developed by the device manufacturer may access certain resources.

Information about the application must be provided by the web runtime, which may be more vulnerable to attack. An advantage of moving policy enforcement into the PZP is that the impact of a malicious web runtime is limited: the proxy may be able to assess the

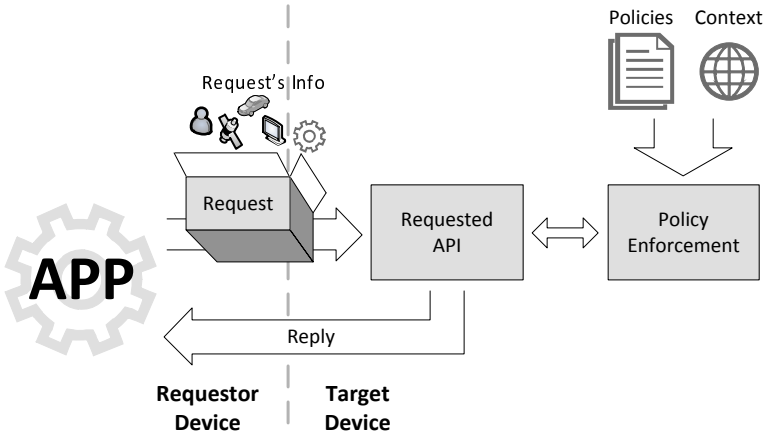


***Figure 4.4:*** *Every application that requires to access a feature sends a request which is filled with information on the subjects and requested feature.*

state of the runtime, and policies applying to all applications will still be enforced.

Figure 4.5 gives an example of a policy which would be held on a local device.

### 4.4.3 Inter-zone policy enforcement

When web applications are running on different devices within a personal zone, access control is managed by the policies defined by the user and distributed on each device. When applications in *different* personal zones are requesting resources (e.g. Alice accesses Bob's location API), it depends on policies created by both users. In this scenario, Bob can create policies referring to Alice in the subject, and therefore control her access to each resource. It can be argued that

```
<policy-set combine="first-matching-target">
  <policy combine="first-applicable">
    <target>
      <subject>
        <subject-match attr="user-id"
            match="U"/>
        <subject-match attr="id" match="A"/>
        <subject-match attr="requestor-id"
            match="R"/>
      </subject>
    </target>
    <rule effect="permit">
      <condition>
        <resource-match attr="api-feature"
            match="http://www.w3.org/ns/
            api-perms/geolocation"/>
      </condition>
    </rule>
    <rule effect="deny">
      <condition>
        <resource-match attr="api-feature"
            match="*"/>
      </condition>
    </rule>
  </policy>
```

```
  <policy combine="first-applicable">
    <rule effect="deny">
      <condition>
        <resource-match attr=
            "api-feature" match="*"/>
      </condition>
    </rule>
  </policy>
</policy-set>
```

***Figure 4.5:*** *Example policy set, showing two policies on a device. The first policy allows user U (using application A on device R) to access geolocation data. The second denies access from every other combination of users, applications and devices.*

other attributes in the subject may be omitted, as Bob has little assurance in Alice's software and may not trust it to correctly identify applications or devices. However, these fields are still present because Alice and Bob may be able to provide such assurances: e.g. they may be in the same physical location, and Bob may know exactly which device Alice is using and whether or not it is trustworthy. To save wasted bandwidth from denied access requests, Alice may chose to implement the more specific aspects of the policy herself. This also makes sense when considering trust: Bob trusts Alice to only let certain applications and devices use the resources he has made available to her. Alice can do this by creating a policy within her personal zone where the resource is an API on Bob's device, but the subjects are her applications and devices. Such a policy would need to be synchronised

between all of Alice's devices.

## 4.5 Conceptual challenges

A first proof-of-concept of the *webinos* system has been specified and implemented. This section will describe three conceptual challenges that have been experienced when designing and developing a cross-domain access control system.

### 4.5.1 Integration with existing security frameworks

Our experiences with *webinos* have added weight to the argument that existing browser access control is inadequate for increasingly sophisticated web applications. Browsers tend to either *always* prevent access to local resources, *prompt* the user for permission, or use implicit consent based on user actions. In a multi-device scenario, however, authorisation prompts cannot always be dictated by just the browser. Some applications require remote access to data and APIs – e.g. remote data wiping features for lost or stolen devices, and 'in case of emergency' applications [1]. These scenarios require more than just authorisation prompts and stored decisions.

Display capabilities on devices are also different: compare a TV to an in-car system, which may be displaying route information. Browser-style access notifications may be difficult to notice or disrupt the experience of the user. The same is true of how users input their ac-

[1] http://ice-app.net/whatis.html

cess control decisions. For some devices, such as sensors or embedded systems, it may even make sense to delegate policy creation to a more capable device. The *webinos* security framework was conceived in order to be flexible enough for all of these settings. Adopting a common policy description language and enforcement framework would allow just interfaces to be adapted on individual devices, rather than the whole authorisation system.

The integration of *webinos* with a browser security model – a sandbox with time-of-use prompting – is in conflict with the *webinos* policy-driven approach. This means that existing browser APIs such as geolocation would need to be changed. We believe this is not necessarily a problem for compatibility, as *webinos* policies can be configured to behave like browsers. For example, access to the geolocation API in the browser results in a user prompt and this decision can be saved per domain. In *webinos*, a policy can be configured to allow or deny access, or prompt the user based on application credentials, user and device context.

### 4.5.2 Subject and resource definitions

In a standard access control scenario, the resources to be protected are unequivocally defined, as is the access requestor. However, in a multi-device scenario different platforms may have different capabilities and implementations, and the importance of related assets to be protected will vary. For example, on a smart phone, access to telephony capabilities must be protected whereas on an in-car system the travelling data may be more important. It is therefore difficult to come up with a definition of high, medium and low risk permissions in the same way

that the Chrome Web Store does.

We assume that each sensitive resource is only exposed by a certain API, reducing the problem to the definition of *webinos* APIs. These had to be specialized in order to cover all the features of each device domain and, conversely, abstract in order to describe common features. For example, many assets are common to all domains (e.g. File APIs), some are common to a subset of sub-domains (e.g. Telephony for in-car systems and smartphone) and some are exclusive to a domain (e.g. TV APIs). The APIs were designed to support domain-specific applications but also to facilitate portability of policies (e.g. a user would like to give an application access to a temporary directory for file system access whatever the target device is or give access to the Telephony API either from his smartphone or car).

As an added complication, the same API can be implemented with different technologies on each platform. The geolocation API can be provided by GPS, Cell-ID on smartphones, IP on laptop, or fixed ADSL line on a TV set. The implication in terms of privacy may vary as a result. For example, a user might grant access to the geolocation API on their PC, as this could only reveal their current home town through IP-based location services. If this permission also revealed their GPS-derived location while in their car or on their mobile device, this could reveal a history of their movements and interactions. The Telephony API is another example: if the end user is roaming on their mobile, then the potential impact of misuse (due to network costs) is greater compared to a home landline. Users might therefore wish to change the authorisation method from 'always allow' to 'prompt first'. Policies should be capable of reflecting which underlying technology is used and the operating mode of the device. As a result, any default

policies will be difficult to define and may quickly become complicated.

### 4.5.3 Policy management with varying ecosystems

Actors, value chains and liability frameworks are different in each device domain. This means that the policy management authority and associated hierarchy can vary from one domain to another as well as the system for provisioning such policies. XACML's policy combining algorithms make it possible to write policies for all of these domains. The use of XACML makes web applications potentially more acceptable in environments where additional security policies may apply. For instance, when an institution's employees may be the target of physical violence (e.g. animal testing laboratories), policies could be created limiting access to their location data from untrusted web applications.

The *webinos* design decision for the first proof-of-concept implementation was to separate policy enforcement from the policy storage and deployment system, leaving each domain to define means and rules for policy deployment and composition of policies from different sources. Again, this implies that each device will need to be shipped with different access control policies and defaults.

### 4.5.4 Shared devices

We have not discussed how to manage devices with multiple owners and users. There are two distinct scenarios. First, devices which have several owners but are only used by one person at a time. For example, a tablet or laptop. In this case multiple personal zone proxies can be

installed on each device, running under different operating system user accounts. This modifies Figure 4.1 to add a 'user account' as a level of indirection between the user and their device.

The second case is more complicated: some devices are used by several people at the same time. For example, a car with several passengers, or a TV in a shared room. At present, *webinos* requires that these devices still belong to one user and be part of one personal zone. However, the user can limit this device so that it has fewer permissions to access the other devices in his or her personal zone. This prevents a guest from using it to access resources on the owner's other devices. We are also investigating linking the policy system to authentication, such that a policy enforcement point on a shared device can demand additional authentication from the end user. This would be OS-specific, for example requiring a short PIN entry using a remote control, and would then make the device capable of accessing other resources and editing local policies. This approach is possible (without policy integration) as part of the defined *authentication API* [49]. However, further features are needed to adapt and control notifications by applications running on shared devices. For example, a shared device might be prevented from displaying notifications about private events, such as receiving a new email.

CHAPTER

# FIVE

# THE M2M USE CASE

## 5.1 Scenario

The improvements achieved in recent years in the fields of miniaturization, connectivity and power consumption are encouraging the spread of pervasive electronic objects which intend to revolutionise the future technological landscape. In the past decade the concept of Internet Of Things has become more and more popular: not only computers, but also objects (things) from our daily life are connected all together and part of the Internet. But, as happened for computers twenty years ago, connecting all things together is not enough: in addition to the connectivity machines have to share also a set of well-known application protocol and paradigms and have to be integrated into the wider figure of the Web. It's the Web of Things (WoT).

The WoT perspective is based upon M2M which is defined as the technology that allows small computing elements, also called *machine*,

to perform specific tasks and communicate or relay information as needed typically over Internet Protocol (IP). WoT so extends M2M concept by adopting consolidate Web technologies to link together several smart objects. In the Web of Things, smart things and their services are fully integrated in the Web by reusing and adapting consolidated technologies and patterns used for traditional Web content. In particular, tiny Web servers are embedded into the objects and the REST technology is applied to resources in the physical world. Using REST, the services exposed on the Web by the smart things usually take the form of a structured XML document or a Javascript Object Notation (JSON) object, which are directly machine-readable. According the WoT vision, in few years household appliances will expose their APIs and they will be reached through Web technologies such as Web Services giving users the ability to create new applications mixing real-world devices doing what is called "physical mashup". As an example an application could switch on a lamp when the room light goes under a certain threshold.

Another typical example of WoT application can be related to e-health. A patient affected by a particular disease can "wear" specific appliances like sensors (blood pressure, glycemic index, etc ) which send clinic information to a doctor who can monitor patients' health. Furthermore doctor could interact with these appliances remotely, for example dosing the correct amount of insulin for the patient.

As mentioned in previous works such as [50], the most important issues to take into account when talking about WoT are how to retrieve things scattered all over the world and how to determine who (user or machine) can access the APIs for a particular smart thing. To meet these needs, some platforms have been proposed [51][52][53] al-

lowing people to connect, use, share and compose Things, services and devices to create personalized applications. Basically these platforms provide a set of APIs to store thing information in a database and to retrieve them. However they don't provide APIs for manipulating the thing. More specifically they don't provide to developers a way to interact with the thing but only a way for read/write information into a database.

Considering the work on cross-platform web application detailed in the previous chapter it is possible to introduce a new approach to WoT with a further step towards the integration of non-human machines in the "well-known" Web; in particular the idea consists on extending the notion of WoT, defining a paradigm similar to the browser (a client on the PC that interprets content and Javascript programs provided by a server) for M2M devices. In other words, as the browser is used by the server as interface to the resources of a PC (display, audio, video, human interface devices - mouse, keyboard, network connectivity) and on top of HTTP(S) the server defines and uses its own communication protocol through AJAX, in this work we present an architecture on which M2M devices and servers communicate with a similar paradigm: instead of a predefined set of capabilities and protocols, M2M devices are provided an execution environment with the same role of the browser (and more precisely Javascript Virtual Machine); servers leverage this execution environment in order to use the resources of M2M devices (sensors and actuators) and to implement over HTTP(S) and AJAX the protocols needed for communication. In order to build such infrastructure the work presented leveraged *node.js*[54] an extended Javascript virtual machine (used in role of browser for M2M devices) and the *webinos* platform which can

be propose as an effective solution that could address some of the most common issues related to WoT.

*webinos* architecture can fit M2M related scenarios, with a simple addition of Javascript API's both for sensors and actuators, which simplify the development of Web applications for an easy interaction with them. Furthermore the *webinos* multi-user management and an access control system ensures flexibility and reliability for the applications. All these aspects suggest *webinos* as a candidate for a secure, user-centric and standard approach to the WoT in order to address the most important requirements of M2M applications.

As an example of addressed M2M scenario we can think a multi-domain application for domotic purpose which allow user to interact with sensors or actuators placed inside her house. For example, user can supervise the state of her sensors (for example temperature, humidity, etc.) directly from her TV; she can also switch on radiators through her smartphone while she is at work or she can lift or lower her garage shutter using her in-car computer.

## 5.2 Related Works

Several works have been presented in literature with the aim of exploiting the web as a mean for standardising the M2M communication. A first class of works focuses on the use of the well-known web services technology taking advantage of their interoperability, platform independence, extensibility, and programming language independence. In particular, authors of [55] introduce a sensor network based on a client-server and publish-subscribe, which provides the ability for preserving

a distributed architecture and enabling the advertising application-specific services. In [56] authors propose to implement web services on the devices natively, so that an intermediate middleware can compose and a large set of low-level basic services to offer higher level services to applications. Some drawbacks of using the web service approach in machine-to-machine infrastructures are discussed in [57], focusing in particular on the overhead and complexity of technologies such as SOAP, XML, and HTTP in constrained environments like energy monitoring, building automation, and asset management.

Other works more explicitly introduce the notion of *Web-of-Things(WoT)* [58], where the integration of web servers directly embedded in the devices is used to avoid internal/proprietary protocols, thus enabling the creation of smart object applications on top Representational State Transfer (REST) architectures. A detailed analysis of several issues involved in the WoT approach is presented in [59]. Further, in [60] use the same approach to design and develop an end-to-end IP based tailored for power consumption optimisation.The idea of exploiting the intelligence and computational power of mobile devices to locally process sensors data is presented in [61].

Other efforts aim at defining and standardizing M2M environments at the API and middleware levels. Authors of [62] present ISIS, an integrated framework providing a complete software tool chain for M2M environments. Pachube [51] provides a RESTful API that allows sensors and smart objects to push data that can be processed later by applications and services. An effort towards standardisation called OpenAPI [63] is developed jointly in Eurescom study P1957 [64] and the EU FP7 SENSEI project. A further research area focuses on the visibility and scalability of environments with a large number of sen-

sors and other connected devices. An identity naming system for all network attached objects using the concept of a globally unique identifier (GUID) is presented in [65] where authors approach the problem by using a distributed hash table to optimize the localization of network objects.

Recently, one of the most active standardisation activity in this area was ETSI TC M2M [66] with the aim of develop and maintain an end-to-end M2M architecture where a RESTful architecture style was adopted. The M2M applications are developed using these resources in order to address different distributed resources on the M2M architecture.

The presented approach, based on *webinos* tries addressing some lacks of the early mentioned works, providing a multi-user access control / privacy policy management system to not only monitoring but also managing, a large set of devices belonging to different domains.

## 5.3 WoT requirements

M2M applications cover an increasing wide variety of segments, addressed devices and environment subtending a multidimention requirements space. Many bodies tried to do a classification of M2M devices and applications [67] [68]: bandwidth requirements, powering and battery duration requirements, life cycle duration, environment, computation power, storage capability were some of the possible indicators used. In reality, the great fragmentation and large number of services suggest to target a flexible architecture and the system should be capable of addressing a large variety of devices and related ap-

plications. A list of high level requirements to be met are analyzed. *Device-Server Protocol and Device resources access*: BACnet, Modbus, SNMP, MQTT, MBus, TCP, UDP are only a subset of transport protocols M2M devices use to communicates with peer servers. Considering also the semantic of transmitted data, we understand how difficult may be the deployment of an E2E horizontal infrastructure to manage a variety of M2M devices and how complex can be the development and management of a M2M server. Looking at the "browser model", once defined that the browser support AJAX [69] (or HTML5 web sockets [70]), it is the server that defines communication protocols and semantics of transmitted data and instructs the device via browser (by mean of downloaded Javascript code) on how and what has to be communicated to the server and which device resources need to be used (e.g. asking user input, display a video, play sounds, get location [71]). The same paradigm is proposed in this work: the M2M device has on board a browser-like application (in majority of M2M application the rendering of HTML content is not needed for the absence of a displays so the simple Javascript Virtual Machine is enough) that behaves as the browser on PC. In this work we inserted in the M2M device a Javascript runtime (practically node.js) and the M2M server is a simple web server from which the device downloads the Javascript code to be executed. The M2M device firmware is agnostic with respect to the server and to the semantic of expected data. To write a M2M application some APIs needs respect to the ample:

- Discovery APIs: to discover the sensors and capabilities available on the M2M device.
- Sensors and Actuators APIs: to access the sensors and actu-

ators present on the M2M device. This APIs abstract all the communication protocols (e.g. I2C, serial ports, GPIO) to be used.

- Device status APIs: to access information on the state of the M2M device (e.g. Network status, Battery status).
- Messaging APIs: to enable the device to remote communication.
- File APIs: to store local information on the M2M device. It should be noticed that this may be substituted by database handling APIs like [72]

In the case of constrained M2M devices (e.g. where it is not possible to host a Javascript engine), the Javascript runtime may reside on a M2M gateway (co-located or in the cloud) interfaced via more simple protocols with the M2M device. In this scenario, the Sensors and Actuators APIs exposed by the M2M gateway to the M2M server will implement the specific protocol to gather data from the remote constraint M2M device. A similar approach is proposed in [73]. *Privacy and access-control*: some M2M data may be sensitive to privacy. Looking to the aforementioned example of a mobile remote health monitoring system, the patient data need to be protected and made available only to authorized parties (e.g. doctor, relatives). Currently the browser access-control mechanism are based on opt-in/opt-out settings and/or on user prompting. This approach may not be used on non-human M2M device where a user is not alway present to allow a certain access or the setting may be too difficult to be defined. The Javascript runtime need to be equipped with a access control system that can interpret policies (eventually dictated by an educated and liable party

in junction with owner of the data e.g medical centre and patient). *Multi-user*: current trends suggest deployment of multi-applications sensors networks. This networks are multipurpose and can be used by different parties for different application. As example, a networks of temperature sensors in a city may be used by weather forecast companies, by pollution control systems, by traffic control. The M2M system need to address multi user capability. With respect to the browser, where usually there is a human "owner" that manages settings, in M2M scenarios the Javascript runtime needs to manage more than one user. *Integrity*: an M2M Device is often not attended and may be exposed to access of unauthorized parties. The Javascript runtime should be able to understand if the device has been tampered. Proper API for attestation may be needed. *Offline applications*: an M2M application may be not connected (partially or frequently). Means to run offline applications are needed. Also modern browsers are supporting offline applications so the same protocols (Offline Web Application [74] or Widgets [75]) may be reused in M2M context.

Considering the above mentioned points, this work leveraged on *webinos*, that defines many means to meet these requirements in an M2M scenario.

## 5.4 Testing Scenario

To demonstrate how *webinos* architecture can be adopted to solve some typical problems of M2M scenarios, we used the *webinos* Sensors API to implement a Web application which allows smartphone users to read values from three kinds of external sensors: tempera-

ture, proximity and light. As described in section 4.3.2, according to the *webinos* approach a user can have multiple devices in his own PZ, and an application installed on a device can access the *webinos* API on an other device. With our demo application an user from his workplace can read on a smartphone the current temperature measured in his house. Figure 6.1 shows the scenario we took into account: the PZ contains a smartphone, a PC, a TV and a *Pandaboard*[76] which represents a *webinos* proxy for the Arduino board.

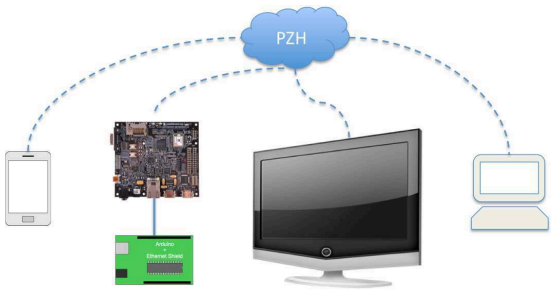


***Figure 5.1:*** *An example of* webinos *scenario*

### 5.4.1 *webinos* Sensors API

As described in section 4.3.2, the *webinos* platform defines Javascript APIs in order to allow Web applications to access device capabilities. Since *webinos* considers each of these capabilities as a service, the Sensors API should be used in combination with the general *webinos* service discovery methods *findServices()* and *bind()*. Using the findServices() method as shown in the following listing, an application can discover all the devices which provide an implementation for the service requested and with the *bind()* method the application chooses

which device to use.

```
//Get list of temperature sensors
//registered in the device through
//the Service Discovery findServices()
//method

findHandle =
 window.webinos.discovery.findServices(
  {api:'http://webinos.org/api/sensors.light'},
  {onFound:sensorFoundCB}
 );
```

Finally the *addEventListener()* method is used by the application to catch sensors' events in order to execute a callback function, for example when a new light value is available. The *webinos.discovery.findServices* requires the name for the feature the developer is asking for, as an example: *http://webinos.org/api/sensors.light* is the feature name if user wants to get access to the light sensor module.

### 5.4.2 *webinos* and Arduino cooperation

In our test we placed an Android smartphone and an Arduino Uno Board into the same Personal Zone.

Devices such as Arduino are considered *dumb devices* because *webinos* doesn't provide an implementation for them, for this reason we connected the Arduino Board to a Pandaboard for which an implementation for the whole *webinos* stack is available. In particular we

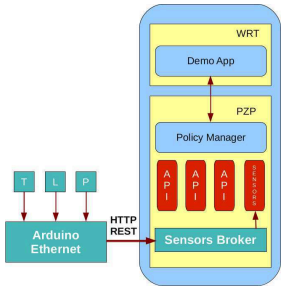


***Figure 5.2:*** webinos *Sensor API Implementation*

implemented a demo Web application which allows users to read values from three kinds of sensors: temperature , proximity and light using *webinos* Sensor API. These sensors are connected to the analog input pins of the Arduino Uno board (from now called simply "board").

For our purpose we equipped the board with an ethernet shield which extends the basic functionalities and introduces the capability of creating a HTTP Web server. The shield provides also a micro SD card slot, so in our demo we used the SD Arduino library to read the configuration parameters for the Web server such as: IP, mac address, refresh rate, etc. Our code written for Arduino allows to define in the configuration file what kind of sensors is attached to the board and to which pin is connected and also to change these parameters avoiding recompilation. As mentioned before, a *webinos* application can be executed on all those platforms (Smartphone, PC, TV, STB) which provide a Web Run Time implementation. For this test we considered *webinos* WRT implementation for a Pandaboard OS host and we connected it with Arduino through an ethernet link. As shown in Figure 5.2 Arduino communicates with the PZP through a broker module which periodically (by default every 60 seconds) receives an

HTTP GET requests from the board containing the current values for the connected sensors and stores them in a data structure. The sensors APIs retrieve these stored values when they are called by the application. The Figurealso shows how the information flow from the sensor APIs to the application goes through the Policy Manager module which ensures the sensor value is read only from those subjects which are allowed by the security policy.

### 5.4.3 The demo application

The demo application was developed using the standard Web technologies: HTML, CSS and the *webinos* Sensors API (Javascript). It allows user to discover those devices, within the PZ, which provide the Sensors service and to subscribe to them. After, sensors values are presented as graphs by mean the Javascript library *Flot*[77]. This library uses jQuery for producing graphical plots of arbitrary datasets, it is simple to use and provides interactive features such as zooming and mouse tracking. The demo application shows a histogram with the current values for light and proximity sensors. For the temperature sensor instead, the application maintains an history of values of last 24 hours.

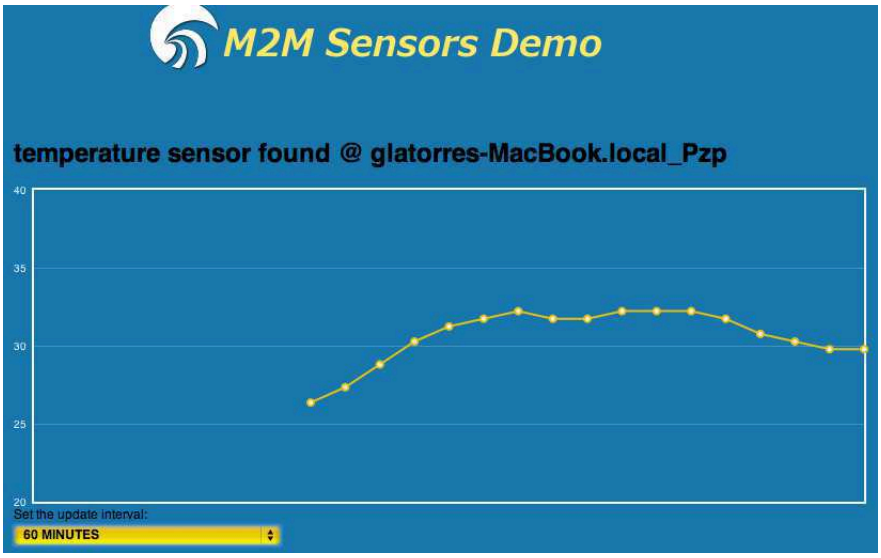


***Figure 5.3:*** *Screenshot for temperature monitoring*

CHAPTER

# SIX

# POLICY MANAGEMENT FOR UGSS

## 6.1 Scenario

Service Oriented Architectures (SOAs) and Web Services are consolidating as the most important technologies for supporting Web application development [78], [79]. Web services provide their APIs publicly, allowing third-party developers to reuse them creating custom Web applications. As an example, it's currently possible to create an application using services such as news feed, weather information, Maps and many others.

One of the most important feature of web services is the possibility to compose them in order to create new services with enriched functionalities. The composition process however, may give rise to some issues related to policy composition. For example, when an application developer assembles existing services which have their own security policies, he must ensure that the resultant policies are not in

conflict with the existing ones. In the past few years, some services on Internet (e.g. Yahoo Pipes, We-Wired Web, MashableLogic) introduced the possibility to create content and services accessible from the web in a simple and reliable way by users for users. The main idea behind these web applications is composing new services starting from the aggregation of data (feeds, web pages) and existing services mixed with user's defined tasks or simply creating new views of the existing content. The result of this composition is commonly referred as User Generated Services or User Generated Content respectively. The large diffusion of everyday's life Internet-capable objects (such as smartphone, tablet and so on) opens new possibilities that make users able not only to generate new contents and services from scratch but also to provide these services to other users. Until a short while ago the main issue in this direction was due to the absence of a common, consistent and easy way to access features offered by these Internet-capable objects. Recently some solutions like Opera Unite[80], Gibraltar[81] and the already mentioned *webinos*[82] arose to cope with these problems and among them *webinos* covers the aspects related to multi-devices scenarios and also offers a rich user's context management supporting as shown access control and data handling policies. For these reasons *webinos* could be considered a potential enabler for User Generated Content or Services.

The interest put on these projects highlights the importance that User Generated Contents and Services might assume in the short term.

In this perspective, there are substantial differences between classic services and User Generated Services, even more when the latter are provided by users through mobile devices. We expect in fact the firsts are practically always available and regulated by permissive ac-

cess control policies which in most cases don't need to be changed frequently by service administrators. This may be different talking about contents or services supplied by users: in social networks, in fact, when users add or remove contents (photos, videos, etc) their policies are updated. Moreover, when users add or remove acquaintances from their list of friends, the number of policy subjects also increase or decrease respectively. In conclusion it is possible to state that, unlike what happens in classical services, policies associated to UGC or UGS may change very frequently due to users' intervention.

In addition, access to a service provided by a user might be allowed or denied on the basis of particular situations in which the user is involved. This can be achieved introducing contextual policies which give the user-provider the opportunity to specify access control rules based on user context information e.g. environment information, social relationships, and so on.

An example of user generated service governed by a context-dependent access control policy is the following:

**Policy example 1** *An user belonging to my group "colleagues" can automatically access my gallery via bluetooth when all the following rules are satisfied*

1. *My battery charge is at least 35%*
2. *I'm at my workplace*
3. *I'm not on the move*

As it's easy to imagine, user's context may change very rapidly and, in a way this implies the continuous change of the current active policy for a device.

In conclusion, User Generated represent a new class of services which substantially differ from traditional services. Many scientific contributions have addressed typical problems related to UGS such as dynamic policy change [10] and context-aware policies [9]. However none of these works has yet taken into account this kind of issues relatively to UGS composition. In this Chapter is shown as composition of services whose access control depends on context-aware policies complicates the policy enforcement on the composed service.

The rest of this Chapter is organized as follows: in Section 6.2 is depicted a realistic scenario which presents problems described so far; Section 6.3 reports the state of the art about user generated services and context-aware policies; Section 6.4 introduces a notation for service composition; in Sections 6.5 and 6.6 a set of problems related to UGS composition are addressed and with the proposal of some approaches to overcome them; finally in Section 6.8 is presented a the proposed cloud approach for UGS composition.

## 6.2 Rationale

The number of electronic devices such as smartphones, tablets, in-car-devices and so on, is going to increase in the next years. According to *webinos*' vision each user will be able to manage communication and interaction among his devices. These devices are virtually contained (registered) inside a user's Personal Zone and might be also shared with others.

Figure 6.1 shows a realistic scenario which consists of two Personal Zones belonging respectively to Alice and Bob. Alice's PZ con-

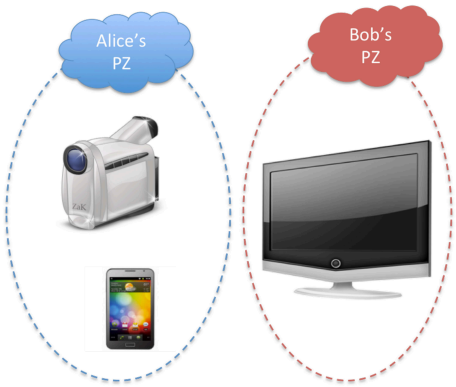


***Figure 6.1:*** *Scenario*

tains a smartphone and a camcorder, Bob's one instead contains a TV. Despite a single device could be "shared" among several Personal Zones, it is owned by a unique user which is responsible for its policies. An example policy for Alice's PZ could be the following:

**Policy example 2**
*Alice's relatives are allowed to see her smartphone's gallery pictures on their TV when she is connected through WiFi.*

Each user can set policies for the devices in his personal zone. In the case that a device is shared among multiple users (and then PZs) and its behaviour is regulated by multiple policies (one for each users) the one written by the owner takes priority over the others.

In the previous section a brief description about UGSs has been introduced. We also pointed out those issues which could arise when services regulated by context-aware policies are composed.

These issues may be experienced for instance considering the scenario depicted in Figure6.2 in which services *Videostream* and *Screen-*

*shot* are provided respectively by Alice and Bob. In particular:

- *Videostream* allows users to receive a video stream from Alice's camcorder.
- *Screenshot* allows users to take snapshots of images reproduced on Bob's TV.

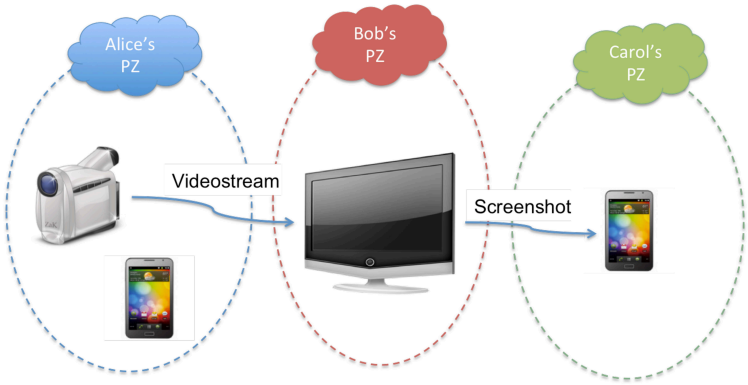


***Figure 6.2:*** *An example of user generated services*

Now let us consider the following example

**Policy example 3** *Alice's policies for Videostream service are*

1. *Alice's parents are always allowed to use Videostream service;*
2. *Alice's relatives are allowed to use Videostream service only when Alice is in Catania;*
3. *Every other subject is not allowed to use Videostream service.*

*Moreover, Bob's policies for Screenshot service are:*

1. *Only Bob's house guests are allowed to use Screenshot service;*

2. *Every other subject is not allowed to use Screenshot service.*

Both Alice's and Bob's policies specify contextual constraints: Alice is willing to offer her service torward her relatives only if she is in Catania (so this policy depends on Alice's context), Bob's policy instead allows only people who are inside Bob's house (so this policy depends on subject's context). Let's imagine that Bob invites Carol (Alice's aunt) at his home to watch a video and suppose that Carol takes some screenshots of this video using her smartphone using TV provided service. When Bob's TV receives a Screenshot request from Carol, it checks its policies and allows Carol's smartphone to access the service, according to Bob's policy. This scenario might raise some problems if Carol try to take screenshots of a content owned by Alice. In this case, even supposing that Bob knows Alice's policy, at the time that he receives a request from Carol he is not able to enforce it because he doesn't know Alice's context information, i.e. location (Is she in Catania?). This contextual is considered "private" information, in fact only Alice knows where she is at a particular moment.

If Bob decides to allow a Carol's request only enforcing his policy (regardless Alice's decision), the following situations will occur:

| # | **Alice's Context** | **Alice's Policy** | **Conflict?** |
|---|---|---|---|
| 1 | A is in Catania | C is A's relative | NO |
| 2 | A is in Catania | C isn't A's relative | YES |
| 3 | A isn't in Catania | C is A's relative | YES |
| 4 | A isn't in Catania | C isn't A's relative | YES |

According to Alice's and Bob's policies, case 1 is the only which doesn't cause any conflicts between Alice's and Bob's authorization decisions.

On the contrary, the conditions considered in cases 2,3 and 4 lead to conflicting situations as Bob doesn't take into account Alice's policy and related context information at the enforcement time.

This is only one of the issues which may arise when multiple User Generated Services are composed. In the rest of the Chapter we will analyze these issues and propose several solutions to solve them.

## 6.3 Related Work

The scientific interest about UGS and UGC fields is growing in these last years. Zhao et al. present a comprehensive survey of current state of art in UGS. They give the specific description of UGS by comparison with the concept of UGC, and then go through different technologies to analysis the challenges of UGS describing advantages and limitations of each approach [83]. Jensen et al. describe some guide lines to support users creation and management of services [84]. Furthermore Tacken et al. investigate the state of the art and the requirements to let the vision of the super prosumer concept become true. They review the current technologies for easy creation and discovery of mobile services and list the identified requirements for user-generated mobile services [85].

Concerning to aspects related to service composition, many works were proposed to prevent conflict behaviours when policies are composed. In his work Satoh et al. propose a security policy composition mechanism for composite services which derives the policy consistency rules and implements the proposed mechanism declaratively to compose security policies semiautomatically from a composite pro-

cess definition and existing security policies of external services [86]. Also Speiser presented an approach that merges an existing policy of a composition with the applicable restrictions into a new policy in order to remove redundancy and inconsistencies [87].

The process of policy composition might be very expensive from a computational perspective and proposed solutions [86] [87] can only be applied if we assume static policies, so that the composition process can be performed only once off-line, when the composite service is made up.

The growing of context-aware services has accelerated the accademic interest toward context-aware policies especially for those scenarios which put mobile devices under the spotlight. Some important contributions in this field have been directed through the definition of policy models suitable for context-aware scenarios [7]. Access control systems should be able to support and understand any new context information in order to address access control requirements. To make this possibile Cheaito et al. presented an extensible access control solution based on XACML making it able to understand new attributes data types including the functions that are used in the policy to evaluate the users requests [8]. Another interesting work is [9] where Li et al. proposed an access control policy model based on context and role that can be appropriate for web service. The model takes context as the center to define and perform access control policies. It uses the contexts of user, environment and resource to execute dynamic roles assignment and constrain the authorization decision. Another interesting work which addresses conflict problems in context-aware policies is [10] where authors propose a framework where authorization for a particular access request is decided dynamically based on

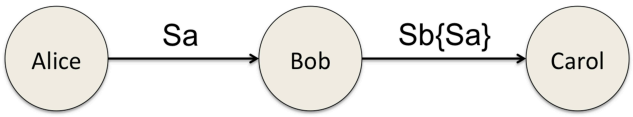


***Figure 6.3:*** *UGSs composition*

context information. They futher support dynamic conflict resolution where current policy is chosen at run time based on context information.

Finally, the emerging of context aware services and relative mobile applications is making it necessary to redesign actual mobile OS' security frameworks. Current smartphone systems allow the user to use only marginally contextual information to specify the behaviour of the applications, Conti et al. propose CRêPE [11], an Android security extension which allows context-related policies to be set (even at runtime) by both the user and authorized third parties locally or remotely. Policies wich can be defined in CRêPE are based on the status of variables sensed by physical (low level) sensors, like time and location.

## 6.4 Service Composition's Notation

Before continuing with the rest of the sections is necessary to define the notation we adopted to describe our case studies. Figure 6.3 shows a graph representation for a user generated service composition where:

- Each node represents a service provider which can be whatever device (smartphone, camera, etc) owned by a user.
- Each edge (for example $A \longrightarrow B$), represents a service provided

by $A$ and requested by $B$.

- Each service provider holds a PolicySet ($PS_a, PS_b, PS_c$) for his service.

We also adopt the notation $S_a$ to state a "component service" provided by an "originator provider" A and the notation $S_b\{S_a\}$ to specify a "composed service", provided by an "intermediate provider" B, which composes $S_a$.

In addition, referring to Figure 6.3, we say that $A$ receives a *direct request* whenever $B$ requires $S_a$, similarly $B$ receives too a direct request each time $C$ requires $S_b\{S_a\}$. We say instead $A$ receives an *indirect request* whenever $C$ requires $S_b\{S_a\}$ from $B$. In the rest of this section we will refer to a PolicySet generically as a policy.

## 6.5 Security policy issues in UGS composition

Web services behavior (either traditional or user generated) is regulated by security access control policies which restrict the access to services from external subjects, represented by applications or other services. In the UGS scenario when a user assembles existing external services he has to guarantee that the policy of the composed service is coherent with policies of all involved external services (also called component services). For example suppose Bob wants to create a new service composing another provided by Alice. When Carol requires access to this new service, Bob can't decide whether to allow or deny Carol only evaluating his policy, but he also should know Alice's de-

cision toward Carol.

For what concerns security in SOA several studies in the literature have addressed this issue by proposing different solutions [86], [87]. They are focused on strategies for static policies' composition and implicitly assume to have all external services' policies. Such strategies may be not appropriate if we consider the case of UGS. There are three main aspects that must be considered in the UGS composition with regard to the access control policy.

- Dynamicity: service related policies may be changed frequently by users.
- Context awareness: policies may depend on users dynamic contextual information (user context).
- Degree of privacy: Users could want to set restrictions on the disclosure degree of their policies towards other users.

Above features characterize UGS composition and give rise to new issues as compared with the traditional service composition.

### 6.5.1 Dynamicity

In fast changing environments in order to promptly react to policies' changes, a security policy infrastructure is required with the following features.

- It should be able to update/synchronize composed services' policies when external services policies' change. For example, if the policy of service $S_a$ changes, it is necessary to synchronize the policies of all composed services which use $S_a$ as external service.

- It should define efficient strategies to ensure that any change in component services' policies does not lead to conflicting authorization decisions taken by the security policy of the composed service.

In this perspective, techniques based on static composition of policies [86] [87] may not be suitable for highly dynamic environments where there is a need to adapt the policies of the composed service dynamically.

### 6.5.2 Context awareness

The use of policies referring to the users' context complicates the authorization decision process to grant access to the composed service. In Policy Example 3 Bob can take a conflict-free decision against a Carol's request only if he knows the information about Alice's context. But this is a remote information as this context is only known by Alice and it depends on the time of Carol's request. Contextual policies therefore raise additional problems in user generated service composition. In particular, the policy manager of the composed service provider needs to know the remote contextual information from which the external services' policies depend on. We refer to this as "context synchronization problem".

### 6.5.3 Degree of privacy

UGS composition can lead to several privacy problems related to policy disclosure. The user (intermediate provider) which assembles a

composed service has to ensure that the associated policy is compliant with all the external services policies.

This is possible if he knows all the policies, that is the behavior of each external service toward all the possible subjects. In many cases, especially in UGS scenario, this could be a serious privacy problem as policies contain private information which the owner of the external service might want to disclose partially (partial privacy) or not at all (full privacy) to other users.

If all external services adopted full privacy on their policies the intermediate service provider would be unable to ensure any compliant authorization decision against a composed service's access request.

Similar problems might arise also in the case of partial privacy. In addition, contextual user information could be in turn under partial or fully privacy restrictions, as it often could represent sensitive user's data.

## 6.6 Security policy strategies in UGS composition

In the following we address the main issues discussed in the previous section and propose a set of possible strategies which may be adopted for policy enforcement of composed UGS. These strategies represent the core on which our cloud approach will be based.

### 6.6.1 Distributed Policy Enforcement

Distributed Policy Enforcement (DPE) is the solution we propose in the case of full privacy degree for the component services' policies. DPE is based on a distributed algorithm for policy enforcement that involves all the providers (partners) which take part in the service composition process. Each partner is characterized by a "not disclosure" policy strategy: this implies that the composed service provider cannot know partner's policies. Using DPE, an authorization decision to deny access to a service is taken only if the individual decisions of all partners have been collected and at least one of them is of type Deny. The PCA algorithm is therefore of type Deny-Override. When a provider receives an access request to its composed service, it firstly evaluates the request locally and if necessary forwards this request to all the external providers involved in the service composition. We can see in detail how DPE works considering Figures 6.3 and 2.1. Suppose Bob is composing a service offered by Alice ($S_a$) in order to provide a new service ($S_b\{S_a\}$). When Carol requires access to $S_b\{S_a\}$, Bob's Policy Enforcement Point (PEP) firstly enforces local policy to decree whether Carol is allowed and if not, it straight rejects the request. On the contrary, if Bobs policy allows Carol, Bob's PEP forwards the incoming request to Alice's PEP (on behalf of Carol). At this stage, Alice's policy manager enforces the local policy and evaluates whether Carol can indirectly access $Sa$. Finally Alice's PEP sends the decision to Bob's one, which issues (or not) the service. As direct consequence of its behavior, DPE is agnostic to dynamic policy changes of external services. Any change in fact is taken into account thanks to the distributed nature of the policy enforcement: each partner takes part

to the final decision on the basis of its policy decision.

DPE works well also in context based policy systems. The authorization decision of each service's partner will take into account in fact also eventual user contextual information. This technique preserves the policy's privacy of all partners and at the same time ensures that each partner decision is not violated.

Summarizing, DPE can be adopted to solve all the issues related to UGS composition we pointed out earlier: dynamic policy changes, context-dependence and policy disclosure. From a practical point of view DPE exhibits a main drawback. DPE can lead to a composed service which could never work due to conflicts between external service's policies. However in this case the problem is not DPE itself but the lack of knowledge about component service's policies. In addition, from a performance point of view some overhead are introduced by DPE every time it is running. When the intermediate service node forwards an access request to the partners it has to transfer to them all the requesters identity information plus the information about the required resources. On the other hand DPE does not introduce any performance degradation due to policy or context synchronization.

### 6.6.2 Local Policy Enforcement

In cases where providers' policies don't contain information potentially subject to privacy issues, these could be stored and evaluated by intermediate providers. Local Policy Enforcement (LPE), unlike DPE, requires policy evaluation only in those service providers which supply a composed service.

This means for instance that an intermediate provider (Bob), in

addition to its PolicySet, must store the component service PolicySet ($PS_a$) and evaluate it whenever he receives requests (such as from Carol).
In general, an intermediate provider who stores all component PolicySets could compose them and then, for each access request, evaluate the composite policy. As already mentioned, the process to derive a composed policy without both conflicts and redundancies from several component policies is quite complex from a computational perspective. For this reason, in a UGS scenario, where policies are dynamic and may change frequently, we can affirm that evaluate them singly is preferable than extracting a composed policy and then evaluating it. So, in our scenario, using LPE, an authorization decision that denies service access to a subject is taken if at least one of the component policy returns a Deny. The PCA algorithm is therefore of type Deny-Override.

Since LPE expects that composed service's providers keep component services' policies, it is necessary to update these every time they change. As shown in Figure 6.4, if $n$ users (providers) requires $S_a$ service in their composition process, each time Alice's policy for $S_a$ changes her policy manager has to synchronize the new policy in all the involved providers. Synchronization could be a burdensome task as a provider (e.g Alice) should maintain information about all the "follower" providers and it is even more problematic if the provider which supplies the service is a mobile device.
In those cases when not only the policy but also the context relative to a service provider may change, it might be not enough knowing these policies to be able to evaluate them. As shown in section 6.2 (policy example 3), though Bob knows that when Alice is in Catania

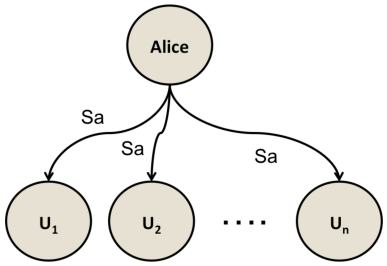


***Figure 6.4:*** *Syncronization problem in LPE*

she is willing to provide Videostream service to Carol, he can't know at the enforcement time where Alice is and thus he can't allow or deny Carol. So, to evaluate a component service's policy an intermediate provider must know the remote context. For this reason we have thought two different approaches to enable policy enforcement by intermediate providers when context-dependent policies are used. The first approach assumes that Bob stores Alice's policyset which contains $n$ context-dependent policies. However, only a subset of these policies is currently active: those related to current Alice's context. So, Alice sends Bob only information about her context and Bob uses this information to understand which of the policies contained in the policy set are active (applicable). This exchange could happen according to two strategies:

(a) Alice tells Bob her context information whenever her context changes;

(b) Bob asks Alice her context information when he receives a request from Carol.

If Alice's context changes frequently it could be better to adopt ap-

proach (b). On the contrary, if Alice's context rarely changes, the proposed variant (a) may be preferable.

The second approach foresees that Bob knows only the current active subset of Alice's policies. In this case Alice sends Bob not context information but straight those policies which her context enables.

In both the two approaches Alice needs to realize when her context changes so as she can advise Bob sending him her actual policy/context information. Therefore there is the need to automatically spot every context changes. For this purpose we propose an extension of XACML architecture (Figure 6.5) which introduces the Context Listener component. In detail, Context Listener reads the policy obtained through the PAP (Policy Administration Point) module and stores both context-dependent parameters and relative boundaries. For example if a policy rule was

*"Permit if battery level is greater than 35%"*

Context Listener will store "*battery_level*" as a parameter and "*greater than 35%*" as the related boundary. Using this information, Context Listener can reorganize user's policy on the basis of his context. In particular, the current values of context-dependent parameters affect the current policy selection. For example the code reported below may represent the logic followed by Context Listener to choose the current policy.

```
if (context_params1 < boundary1)
  current_policy = P1
if (context_params2 > boundary2)
  current_policy = P2
```

So, Context Listener periodically polls the PIP (Policy Information Point) module to obtain actual values for the contextual parameters (such as current battery level) and checks whether these values remain within the boundaries defined by the policy. When at least one of the parameters included in the rule exceeds its boundary, the Context Listener communicates to interested intermediate providers the current user's context or policy.

In summary, LPE can be adopted if there aren't any privacy concerns, but the synchronization process requires an overhead that as we will show in section 6.7 may be not acceptable in case of devices with reduced performances as for example mobile phones.

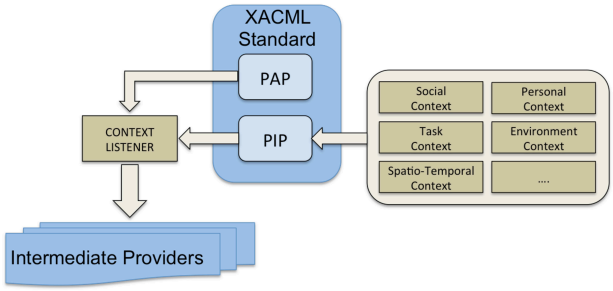


***Figure 6.5:*** *XACML extension with Context Listener*

### 6.6.3 Policy's Context Obfuscation

Local Policy Enforcement requires distribution of policies or parts of them. The process to exchange this kind of data requires some mechanisms to assure content authenticity and avoid tampering. Privacy is the other main aspect to cope with in situations such as this one, where personal information (i.e. context data) are exchanged. In fact,

the policies themselves are privacy sensitive information because they can contain users' personal choices (i.e. rules that are be applied to specific subjects) and may depend on context parameters (e.g. users' location details). Policy Information Points are allowed to elicit, from remote providers (i.e. users' devices), personal context data in order to allow policy enforcement. This means that knowing user's policy and the relative decision toward a request, it could be possible to retrieve information about user's context. This obviously leads to privacy problems.

In literature the problem of policy's content protection is faced in different works and for example in [88] where the authors propose techniques based on encryption. Our proposal rely on obfuscation methods in order to mask sensitive information sending the policy structure in plain text.

Suppose the policy below is that chosen by Alice for videostream service.

Since this policy depends on Alice's context (specifically on Alice's location), giving it in plain-text to Bob could give rise to privacy issues: Bob in fact, knowing the policy, could realize that Alice is at home at the instant when Carol is allowed to get videostream service. This information does not concern Bob: he only should know that Carol will be able to get videostream service (indirectly through screenshot service) when a certain Alice's context will be active, but he should not know which this context is. The policy below is that stored by Bob: it is similar to Alice's policy but contains encrypted context's information (params and values). So, using LPE Alice will transfer to Bob the pair $< E67QF1, AR7MG6 >$ representing her context, instead of the private information $< location, home >$.

```
<policy-set id="Alice's policyset">
  <policy algorithm="permit-overrides">
    <target>
      <subject param="cert" match="Carol.cert"/>
      <resource param="service" match="videostream"/>
    </target>
    <rule id="5ABE" effect="permit">
      <condition>
        <context param="location" match="home"/>
      </condition>
    </rule>
    <rule effect="deny"/>
  </policy>
  <policy>
    <target>
      <subject id="F3MK"/>
    </target>
    <rule effect="deny"/>
  </policy>
</policy-set>
```

```
<policy-set id="Obfuscated Alice's policyset">
  <policy algorithm="permit-overrides">
    <target>
      <subject param="cert" match="Carol.cert"/>
      <resource param="service" match="videostream"/>
    </target>
    <rule id="5ABE" effect="permit" obfuscated>
      <condition>
        <context param="E67QF1" match="AR7MG6"/>
      </condition>
    </rule>
    <rule effect="deny"/>
  </policy>
  <policy algorithm="deny-overrides">
    <target>
      <subject id="F3MK" obfuscated/>
    </target>
    <rule effect="deny"/>
  </policy>
</policy-set>
```

### 6.6.4 Partial Disclosure Policy Enforcement

Describing DPE and LPE we pointed out some advantages and disadvantages of both solutions. DPE solves problems related to dynamic policy/context changes and privacy issues related to policy disclosure, but it could be very difficult to generate a new service without any knowledge about policies related to component services. This in fact may lead to have composed services which do not work properly: for instance, if Bob decides to provide $S_b\{S_a\}$ only after *5:00 pm* without knowing that Alice is willing to provide $S_a$ only before *3:00 pm*, his service will never work. On the other hand, with LPE approach, intermediate providers know component policies: this could involve privacy problems (discussed in previous sections), but it represents the best way, for an intermediate provider, to supply a working service. For these reasons we want to introduce a new approach which can be considered a somewhere in between DPE and LPE: we call it Partial Disclosure Policy Enforcement (PDPE).

According to the idea behind PDPE, when an intermediate provider wants to supply a composed services to a set of known subjects, he can obtain from the component provider not the whole policy but only the information regarding these subjects. PDPE thus provides two phases:

1. an *initialisation* phase where an intermediate provider (Bob) can "negotiate" the policy disclosure with the component provider (Alice) requiring only the policy's information toward a predetermined set of subjects.

2. an *incremental update* phase where the intermediate provider

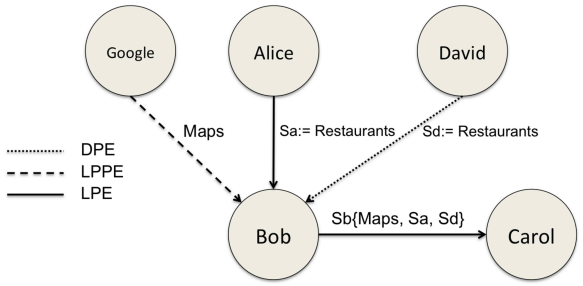


***Figure 6.6:*** *An hybrid composition scenario*

requires additional policy's information every time a new subject (which is not included in the original set of the considered subjects) requires access to the service.

The first phase can be optional as an intermediate provider could not negotiate the policy disclosure for a first set of subjects but build this set whenever a subject requires the service. For example, suppose Bob doesn't know Alice's policy for $S_a$. When Carol requires $S_b\{S_a\}$ Bob informs Alice who sends back to Bob a piece of her policy regarding Carol. From that moment onwards, Bob stores this partial policy, enforcing it every time Carol requires $S_b\{S_a\}$ service. Whenever a new subject (for example David) requires $S_b\{S_a\}$, Bob repeats the procedure, and so on.

Since even partial policies may depend on user's contextual information, PDPE employs the same approaches shown in Figure 6.5 to overcome problems related to policy/context changes. In addition it might apply the obfuscation technique discussed in previously.

### 6.6.5 Considerations

The proposed approaches mainly differ in the amount of policy's information revealed by component to intermediate providers. However all the discussed approaches can be combined to meet different users' security requirements. Suppose to have a scenario where Bob wants to provide a service which shows a Google Map decorated with his and his friends' preferred restaurants. This scenario is represented in Figure 6.6. Alice and David provide two services ($S_a$ and $S_d$) which give information about their preferred restaurants. We can note that all the considered approaches are employed: Google, using PDPE, gives Alice only a piece of its policy (only the parts related to those subjects that Bob wants to consider), Alice indeed, doesn't bother about privacy issues and gives Bob her whole policy (LPE), finally David doesn't want that Bob knows his policy, so he adopts DPE.

## 6.7 Mobile environments constraints

As aforementioned the broad diffusion of smart mobile devices combined with the presence of platforms able to simplify interactions among them, represent a good starting point for the creation of UGCs and UGSs. In some cases these services could strictly depend on devices' features and local context. If we consider, for example, the scenario depicted in section 6.2 we can find out that the service created by Alice is provided thanks to the presence and the features of her smart device. This scenario is just one example that points out as the next step is represented by those services that are not only generated but also provided by users through their devices (User Provided

Services or UPSs). Limitations may be due to the utilisation of kind of devices which are not supposed to work as providers. This is the case of smartphones and handsets in general. In fact, it is possible recognizing problems of at least three categories which are performance, scalability and presence. For what concerns performances it is clear that providing a service that is resource consuming (for example in terms of CPU time, memory and bandwidth) may reduce the usability of devices' main functionalities. Scalability issues are related to the number of users which can concurrently access our self-provided service.

The last point regards presence: some devices could be not reachable in any moment due to a temporary lack of connectivity (for example a car or a smartphone inside a gallery or in a place without coverage) or just because they are running out of battery. As known devices might provide services or contents and can regulate access to them via context-based policies. In the first case the device presence is obviously needed to guarantee the service availability and in the second case it is needed to gather the context information without which the policy to access or handle the remote content cannot be enforced.

It is also important notice that the problems described in the previous sections still remain. In fact, it is difficult to properly handle policy disclosure and cope with the synchronization process of context and policy, especially in the case of many actors.

## 6.8 Service mediator cloud approach

In order to cope with the issues presented above, we propose another structure that differs from the one depicted in Figure 6.3 in a component defined as *service mediator*. The structure of this architecture is shown in Figure 6.7.

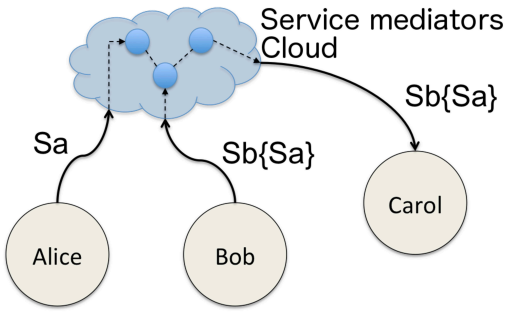


***Figure 6.7:*** *Proposed cloud architecture*

The new structure based on cloud still allows usage of the already described approaches e.g. LPE, DPE or their combinations. It is important to point out that in the case of the cloud based solution the policy is enforced by cloud components and not by intermediate providers. We can refer to this approach as Cloud Policy Enforcement (CPE).

The role of the *service mediator*, that is intended to be a cloud component, is manifold, in fact it is possible to recognize at least the following uses:

(a) Synchronize user's preferences, context information and policies

(b) Resolve problems related to "device presence"

(c) Reduce disclosure of sensitive data, context information and policies

(d) Provide unified service discovery

(e) Integrate logic to switch among different policy and context exchange approaches

(f) Enhance scalability and performance

One of the main advantages of a cloud approach regards synchronization (a). In fact, compared to the peer-to-peer model presented before, a cloud based architecture simplifies the process of exchanging and keeping up to date all the information needed. This might be done for example through a publish/subscribe notification system.

Furthermore a such publish/subscribe mechanism could be useful to resolve presence related issues (b), enabling users and devices to leave online records of taken decisions, context information and other data also when they are disconnected.

This problem, as mentioned before, is very relevant in the case of mobile devices. If we consider for example the scenario presented in section 6.2 adding the clause that Alice allows Bob to record her videostream and reuse it with the aforementioned restrictions, the knowledge of Alice's location becomes very important. For obvious reasons is not possible to know Alice's location at any moment (e.g. all her devices run out battery) but with the cloud approach here presented it is possible to communicate and preserve context information e.g. the last location available, for a more accurate evaluation.

This context recording mechanism could be implemented also in a non cloud system but with some issues. For example it is necessary to

distribute context information to other nodes (disclosure of potentially sensitive data) and in an approach like LPE, the number of nodes that should be notified by the service provider at each context change may be very high.

On the contrary with the cloud approach it is possible to limit disclosure of personal data (c). In fact the cloud component could be the only point where sensitive information are stored. In addition it enforces remote access requests on behalf of the remote service provider with the advantage of limiting exchange of policies, context information and sensitive data.
Devices can expose services registering them in the cloud. There the *service mediator* can: make them discoverable (d), manage their status and as aforesaid regulate remote access to them through policies.

The *service mediator* could also contain some logic in order to switch between approaches like DPE and LPE (e). This smart switching system can allow users to change the desired degree of privacy at run time.

The gain in terms of scalability and performances (f) is due to the fact that the *service provider* will not be directly contacted by clients because exposed services will be accessed by the *service mediator*. It will be able to multiplex a service or serialize access to it in the case he cannot directly provide that service. Considering again the videostream service, it is easy to imagine that a solution like the one proposed - with a server that acquires the videostream (*service provider*) and multiplexes it to other consumers - can reduce the provider's workload.
To test the cloud structure here presented we need to implement a prototype that at least enables users to define for some personal devices a

set of exposed features and services, policy and preferences related to them, context information available. This prototype has also to cope with synchronization and security problems described here.

As mentioned before the project *webinos* allows users to expose their devices' features to other devices registered in a virtual area called Personal Zone (PZ). The PZ has the main role of providing in a simple manner access to local and remote services. It creates also an abstraction from underlying communication technologies.

*webinos* leaves open the possibility for applications to expose also some functionalities accessible by others applications/devices. *webinos*' approach consists of an architecture, depicted below Figure 6.8, that at an high level contains a component called Personal Zone Hub (PZH) which corresponds to the *service mediator* introduced above and the component called Personal Zone Proxy (PZP).

As already stated before, the PZH is the element used to create the personal zone and enable communication between nodes of the same or a different PZ. It allows the discovery of other PZHs and manages the synchronization process (e.g. of user's preferences, policy and context data). A PZH manages also security aspects. In fact it is the certificate authority that issues certificates for each PZP registered in a PZ. These certificates allow mutually authentication and also encryption of the messages exchanged among devices in the zone.
The other main component of *webinos* architecture is the PZP that runs on each *webinos enabled device* and manages the communication between the device and the main hub (PZH). The PZP also stores all the information that should be synchronized with the hub in the case it is not reachable in order to start the synchronization process when the PZH becomes again available. The PZP is responsible for

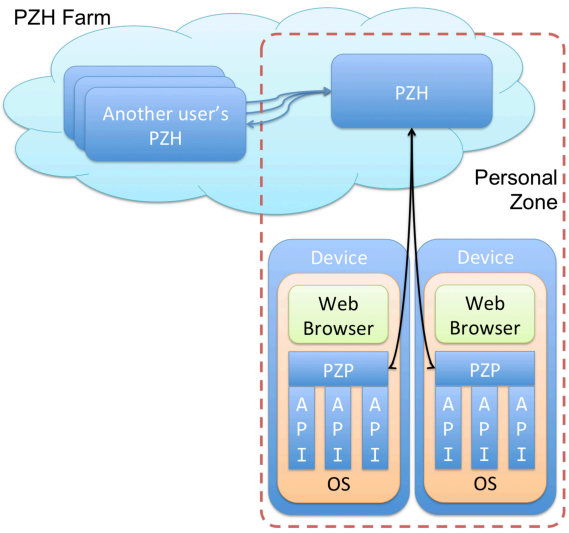


***Figure 6.8:*** webinos *architecture*

the local discovery of features/services accessible through a set of API which exposes some capabilities of the underlying operating system. In *webinos*, applications are widgets, compliant to W3C standard [89], written in Javascript, HTML, and CSS. They run in a web browser or in a custom widget runtime and can access the local/remote features through the PZP.

From this description it is possible notice that the structure proposed by *webinos* well suits to the cloud approach where the *service mediator* is an extension of the PZH able to manage not only security and synchronization aspects but also service multiplexing. The use of *webinos* as a basis for our work make possible to move the focus on privacy/security issues. It remove the necessity of implementing a new system from the scratch to synchronize data like policies, context

information and user's preferences.

CHAPTER

# SEVEN

# CONCLUSIONS AND FUTURE WORK

This dissertation presented some models for security and privacy policy management based on XACML. Access control issues are addressed in both single- and multiple- device scenarios. In single-device scenarios (described in Chapter 3, in which only Android handsets were taken into account, the effort was focused on extending the access control model of the underlying operating system. The extension proposed, named *SecureDroid*, enables policy enforcement not only while an application is going to be installed (which is the default solution on Android systems) but also afterwards, during its run-time. Users can take advantage of this flexible strategy choosing, for example, different behaviours depending on many factors such as context-based information. Recently, Google decided to move toward a more fine grained policy management system: the company released in July 2013, within Android 4.3, an hidden feature called "App Ops" that lets a user to selectively disable some permissions for specific appli-

cations.[1] In Chapter 3, is also described as *SecureDroid* compared to other security frameworks (i.e. Apex and CRêPE) has the advantage of supporting context-based information to determine policy parameters (not supported by Apex) and acting only after the security check made by Android in order to reduce computation and energy consumption (due to continuous access to sensors reads) if compared to CRêPE.

In multiple-device scenarios as such as those cross-platform multi-domain supported by *webinos*, the policy management architecture described in Chapter 4, makes a number of significant changes from the current state of the art, showing policies which are both abstract enough to be transferable to different devices, as well as specific to particular scenarios. The experience in the first implementation of the platform has highlighted several challenges, such as the different implementation of common APIs and how to manage shared devices. *webinos* also exported the Web/Javascript based browser model and the secure, multi-user and cross-domain approach to the WoT, demonstrating how the interoperability through the use of spread technologies (such as those related to the Web) is possible. The proposed policy management has proven to be useful also in systems characterised by the presence of User Generated Services and User Provided Services. These kind of services controlled by users have an high level of dynamicity often related to the user's context. They created new challenges as shown in Chapter 6.

Given these conclusions, further work could be focused on putting the user in the loop of policy management, also to let him understand-

[1] http://www.androidpolice.com/2013/07/25/app-ops-android-4-3s-hidden-app-permission-manager-control-permissions-for-individual-apps/

ing why it is so important. This can be done in different steps: initially, for example, studying an effective and easy-to-use system for delegation, in a way that a user can ask e.g. friends, forums, user support groups, professionals or trusted service providers for help on managing his policies and learn from this process. Another step could be done in the direction of better designed policy management UIs: operations like policy editing or interaction with contextual data should be quick and easy and this can be realised only weighting complexity and expressivity to find the proper trade-off between them. In the context of the interaction user / policy management system, further work can be done on notification mechanism that if properly designed can enhance user experience and can be taylored to different device types.

In addition, it is of primarily importance studying and implementing tools to support developers in writing appropriate permissions and designing applications. In particular, allowing developers to trial their applications based on typical user policies (perhaps gained through user groups or anonymous feedback) would help identify where too few or too many permissions were requested, as well as where applications do not fail gracefully.

# BIBLIOGRAPHY

[1] S. Monteleone, "Contributed code repositories: https://github.com/smonteleone," 2013.

[2] webinos Project Team, "webinos repositories: https://github.com/webinos," 2013.

[3] webinos Project Team, "webinos apps repositories: https://github.com/webinos-apps," 2013.

[4] T. Moses, "extensible access control markup language 2.0 specification set." http://www.oasis-open.org, Feb 2005.

[5] K. Benton, L. J. Camp, and V. Garg, "Studying the effectiveness of android application permissions requests," in *Fifth International Workshop on SECurity and SOCial Networking*, 2013.

[6] G. Thomson, "Byod: enabling the chaos," *Network Security*, vol. 2012, no. 2, pp. 5–8, 2012.

[7] H. Yahyaoui and M. Almulla, "Context-based specification of web service policies using wspl," in *Digital Information Management*

*(ICDIM), 2010 Fifth International Conference on*, pp. 496 –501, july 2010.

[8] M. Cheaito, R. Laborde, F. Barrere, and A. Benzekri, "An extensible xacml authorization decision engine for context aware applications," in *Pervasive Computing (JCPC), 2009 Joint Conferences on*, pp. 377 –382, dec. 2009.

[9] H. Li, Y. Yang, Z. He, and G. Hu, "Context-aware access control policy research for web service," in *Instrumentation, Measurement, Computer, Communication and Control, 2011 First International Conference on*, pp. 529–532, oct. 2011.

[10] A. Mohan and D. M. Blough, "An attribute-based authorization policy framework with dynamic conflict resolution," in *Proceedings of the 9th Symposium on Identity and Trust on the Internet*, IDTRUST '10, (New York, NY, USA), pp. 37–50, ACM, 2010.

[11] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Crêpe: A system for enforcing fine-grained context-related policies on android," *Information Forensics and Security, IEEE Transactions on*, vol. 7, no. 5, pp. 1426–1438, 2012.

[12] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, (New York, NY, USA), pp. 328–332, ACM, 2010.

[13] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings*

*of the FREENIX Track: 2001 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2001.

[14] J. Lyle, S. Monteleone, S. Faily, D. Patti, and F. Ricciato, "Cross-platform access control for mobile web applications," in *Policies for Distributed Systems and Networks (POLICY), 2012 IEEE International Symposium on*, pp. 37–44, 2012.

[15] A. Taivalsaari and T. Mikkonen, "The web as an application platform: The saga continues," in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pp. 170 –174, September 2011.

[16] "Geolocation API Specification: W3C Editor's Draft." `http://dev.w3.org/geo/api/spec-source.html`, February 2010.

[17] "Device APIs Working Group Website." `http://www.w3.org/2009/dap/`, December 2011.

[18] "Bmw connecteddrive," April 2012.

[19] L. Jedrzejczyk, B. A. Price, A. K. Bandara, and B. Nuseibeh, "On the impact of real-time feedback on users' behaviour in mobile location-sharing applications," in *Proceedings of SOUPS '10*, pp. 14:1–14:12, ACM, 2010.

[20] The W3C, "Mobile Web Application Best Practices," December 2010.

[21] The Wholesale Application Community, "Glossary of terms," August 2011.

[22] M. Zalewski, *The Tangled Web: A Guide to Securing Model Web Applications*. No Starch Press, 2011.

[23] "Widget Access Request Policy: W3C Recommendation," February 2012.

[24] W. Enck, M. Ongtang, P. McDaniel, W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," *Security Privacy, IEEE*, vol. 7, pp. 50 –57, Jan - Feb 2009.

[25] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2011.

[26] A. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mock-droid: trading privacy for application functionality on smartphones," in *Proceedings of HotMobile 2011*, 2011.

[27] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pp. 328–332, 2010.

[28] M. Conti, V. Nguyen, and B. Crispo, "Crepe: Context-related policy enforcement for android," in *Information Security* (M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic, eds.), vol. 6531 of *Lecture Notes in Computer Science*, pp. 331–345, Springer Berlin / Heidelberg, 2011.

[29] N. Reddy, J. Jeon, J. A. Vaughan, T. Millstein, and J. S. Foster, "Application-centric security policies on unmodified android," Tech. Rep. 110017, UCLA Computer Science Department, 2011.

[30] "SEAndroid Website." `http://selinuxproject.org/page/SEAndroid`, March 2012.

[31] "WAC Specifications 2.1." `http://specs.wacapps.net/`, January 2012.

[32] S. Godik and T. Moses, "EXtensible Access Control Markup Language (XACML) version 1.1." `http://www.oasis-open.org`, May 2005.

[33] "Primelife policy language." `http://www.primelife.eu/results/documents/153-534d`, August 2010.

[34] Claudio A. Ardagna et al., "Advances in access control policies," in *Privacy and identity management for life*, pp. 327–341, Springer, 2011.

[35] D. Lin and A. Squicciarini, "Data protection models for service provisioning in the cloud," in *Proceedings of SACMAT*, pp. 183–192, ACM, 2010.

[36] Y.-G. Kim, C.-J. Mon, D. Jeong, J.-O. Lee, C.-Y. Song, and D.-K. Baik, "Context-aware access control mechanism for ubiquitous applications," in *Advances in Web Intelligence*, vol. 3528 of *Lecture Notes in Computer Science*, pp. 932–935, Springer Berlin / Heidelberg, 2005.

[37] A. Corradi, R. Montanari, and D. Tibaldi, "Context-based access control management in ubiquitous environments," in *Network Computing and Applications, 2004. (NCA 2004). Proceedings. Third IEEE International Symposium on*, pp. 253 – 260, aug.-1 sept. 2004.

[38] Mazurek et al., "Access control for home data sharing: Attitudes, needs and practices," in *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, (New York, NY, USA), pp. 645–654, ACM, 2010.

[39] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, pp. 263–277, June 2007.

[40] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The multi-principal os construction of the gazelle web browser," in *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, (Berkeley, CA, USA), pp. 417–432, USENIX Association, 2009.

[41] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A safety-oriented platform for web applications," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, (Washington, DC, USA), pp. 350–364, IEEE Computer Society, 2006.

[42] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the incoherencies in web browser access control policies," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 463 –478, may 2010.

[43] C. Ai, J. Liu, C. Fan, X. Zhang, and J. Zou, "Enhancing personal information security on android with a new synchronization scheme," in *Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on*, pp. 1 –4, sept. 2011.

[44] The webinos consortium, "User expectations of privacy and security," March 2011.

[45] The webinos consortium, "Use cases and scenarios," March 2011.

[46] E. R. B. Parducci, H. Lockhart, "XACML v3.0 Privacy Policy Profile Version 1.0." `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-privacy-v1-spec-cd-03-en.pdf`, March 2010.

[47] C. A. Ardagna, S. De Capitani di Vimercati, S. Paraboschi, E. Pedrini, and P. Samarati, "An xacml-based privacy-centered access control system," in *Proceedings of the first ACM workshop on Information security governance*, WISG '09, pp. 49–58, ACM, 2009.

[48] "Bondi architecture and security requirements appendices." `http://bondi.omtp.org/1.01/security/BONDI_Architecture_and_Security_Appendices_v1_01.pdf`, July 2009.

[49] The webinos consortium, "Phase 1 device, network and server-side api specifications," November 2011.

[50] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the internet of things to the web of things: Resource oriented architecture and best practices," 2011.

[51] "Pachube web site." `https://cosm.com/`.

[52] "Evrythng web site." `https://evrythng.net/`.

[53] "Paraimpu web site." `http://paraimpu.crs4.it/`.

[54] "nodejs." `http://www.nodejs.org`, 2011.

[55] E. Sadok and R. Liscano, "A web-services framework for 1451 sensor networks," in *Instrumentation and Measurement Technology Conference, 2005. IMTC 2005. Proceedings of the IEEE*, vol. 1, pp. 554 –559, may 2005.

[56] D. Savio and S. Karnouskos, "Web-service enabledwireless sensors in soa environments," in *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pp. 952 –958, sept. 2008.

[57] Z. Shelby, "Embedded web services," *Wireless Communications, IEEE*, vol. 17, pp. 52 –57, december 2010.

[58] D. Guinard and V. Trifa, "Towards the web of things: Web mashups for embedded devices," in *In 2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009)*, 2009.

[59] E. W. D. Guinard, V. Trifa, "A resource oriented architecture for the web of things," 2010.

[60] N. D. C. W. Colitti, K. Steenhaut, "ntegrating wireless sensor networks with the web," in *IPSN 2011*, 2011.

[61] D. Trossen and D. Pavel, "Building a ubiquitous platform for remote sensing using smartphones," in *Mobile and Ubiquitous Systems: Networking and Services, 2005. MobiQuitous 2005. The Second Annual International Conference on*, pp. 485 – 489, july 2005.

[62] X. Su, R. Svendsen, H. Castejon, E. Berg, and J. Zoric, "Towards an integrated solution to internet of things - a technical and economical proposal," in *Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference on*, pp. 46 –51, oct. 2011.

[63] "Openapi." `http://www.openbloomberg.com/open-api/`, 2011.

[64] "Eurescom project p1957, open api for m2m applications." `http://www.eurescom.de/public/projects/P1900-series/P1957/`, 2011.

[65] F. Andreini, F. Crisciani, C. Cicconetti, and R. Mambrini, "A scalable architecture for geo-localized service access in smart cities," in *Future Network Mobile Summit (FutureNetw), 2011*, pp. 1 –8, june 2011.

[66] "Draft etsi ts 102 690 v0.13.3 (2011-07) technical specification." `http://www.etsi.org`, 2011.

[67] O. M. Alliance, "White paper on m2m device classification," 2012.

[68] 3rd Generation Partnership Project, "Technical specification group services and system aspects;service requirements for machine-type communications." http://www.3gpp.org/ftp/Specs/html-info/22368.htm, 2011.

[69] W3C, "Xmlhttprequest level 2." http://www.w3.org/TR/XMLHttpRequest/, 2012.

[70] W3C, "The websocket api." http://www.w3.org/TR/websockets/, 2012.

[71] W3C, "Geolocation api specification." http://www.w3.org/TR/geolocation-API/, 2012.

[72] W3C, "Web storage." http://www.w3.org/TR/webstorage/, 2011.

[73] O. M. Alliance, "Enabler release definition for gateway management object," 2012.

[74] W3C, "Offline web applications." http://www.w3.org/TR/offline-webapps/, 2011.

[75] W3C, "Widget packaging and xml configuration." http://www.w3.org/TR/widgets/, 2011.

[76] "Pandaboard web site." http://pandaboard.org.

[77] "Flot, attractive javascript plotting for jquery." http://code.google.com/p/flot/.

[78] A. J. Lee, J. P. Boyer, L. E. Olson, and C. A. Gunter, "Defeasible security policy composition for web services," in *Proceedings of the fourth ACM workshop on Formal methods in security*, FMSE '06, (New York, NY, USA), pp. 45–54, ACM, 2006.

[79] R. Pessoa, E. Silva, M. van Sinderen, D. Quartel, and L. Pires, "Enterprise interoperability with soa: a survey of service composition approaches," in *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*, pp. 238 –251, sept. 2008.

[80] H. S. Tømmerholt, "Opera Unite developer's primer." `http://dev.opera.com/articles/view/opera-unite-developer-primer-revisited/`, Oct 2009.

[81] K. Lin, D. Chu, J. Mickens, L. Zhuang, F. Zhao, and J. Qiu, "Gibraltar: Exposing hardware devices to web pages using ajax," in *Proceedings of WebApps, USENIX*, jun. 2012.

[82] C. Fuhrhop, J. Lyle, and S. Faily, "The webinos project," in *Proceedings of the 21st international conference companion on World Wide Web. WWW '12 Companion, New York, NY, USA*, pp. 259–262, 2012.

[83] Z. Zhao, N. Laga, and N. Crespi, "A survey of user generated service," in *Network Infrastructure and Digital Content, 2009. IC-NIDC 2009. IEEE International Conference on*, pp. 241 –246, Nov. 2009.

[84] C. S. Jensen, C. R. Vicente, and R. Wind, "User-generated content: The case for mobile services," *Computer*, vol. 41, pp. 116–118, Dec. 2008.

[85] J. Tacken, S. Flake, F. Golatowski, S. Prüter, C. Rust, A. Chapko, and A. Emrich, "Towards a platform for user-generated mobile services," in *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*, pp. 532–538, april 2010.

[86] F. Satoh and T. Tokuda, "Security policy composition for composite web services," *Services Computing, IEEE Transactions on*, vol. 4, pp. 314 –327, Oct.-Dec. 2011.

[87] S. Speiser, "Policy of composition ? composition of policies," in *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pp. 121–124, june 2011.

[88] C. Dong, G. Russello, and N. Dulay, "Shared and searchable encrypted data for untrusted servers," in *Data and Applications Security XXII* (V. Atluri, ed.), vol. 5094 of *Lecture Notes in Computer Science*, pp. 127–143, Springer Berlin Heidelberg, 2008.

[89] "Widget interface." `http://www.w3.org/TR/widgets-apis/`, May 2012.