Model-Driven Configuration of SELinux Policies

Berthold Agreiter and Ruth Breu

University of Innsbruck, Institute for Computer Science
A-6020 Innsbruck, Technikerstr. 21a
{berthold.agreiter,ruth.breu}@uibk.ac.at

Abstract. The need for access control in computer systems is inherent. However, the complexity to configure such systems is constantly increasing which affects the overall security of a system negatively. We think that it is important to define security requirements on a non-technical level while taking the application domain into respect in order to have a clear and separated view on security configuration (i.e. unblurred by technical details). On the other hand, security functionality *has to be tightly integrated* with the system and its development process in order to provide comprehensive means of enforcement. In this paper, we propose a systematic approach based on model-driven security configuration to leverage existing operating system security mechanisms (SELinux) for realising access control. We use UML models and develop a UML profile to satisfy these needs. Our goal is to exploit a comprehensive protection mechanism while rendering its security policy manageable by a domain specialist.

1 Introduction

The pervasiveness of computing is constantly increasing. Its penetration does not stop for highly critical applications, it is rather increasing (prime examples being electronic medical records, telemedicine and electronic voting). Many applications even share the same security requirements, e.g. data may only be read by specific users or certain users are not allowed to modify data in a database.

Furthermore, the level of protection also depends on the number of possible ways to access data. E.g., when patient information is stored in plaintext but only the editing application implements access control mechanisms, protection is very low since an attacker is not forced to use this specific application. For a comprehensive protection, all possible ways need to be considered to allow usage only in desired ways, i.e. not only specific target applications need to be constrained but also their environment.

This motivates the need for *complete mediation*. The overall goal is to protect data so that it can only be accessed under certain circumstances [20]. SELinux is an extension to the Linux kernel enforcing *Mandatory Access Control* (MAC) [15] on kernel objects (i.e. files, sockets, descriptors, fifos etc.) upon each access attempt. This means, its access control checks on system calls can not be bypassed. It provides a centrally administered policy with security rules which

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part II, LNCS 5871, pp. 887–904, 2009.
© Springer-Verlag Berlin Heidelberg 2009

must be followed by any subject. These rules may only be altered by a central authority.

However, this poses a new challenge namely that the configuration of comprehensive security policies tends to be very complex. This is especially true for SELinux since its Reference Policy already defines over 50 object classes with up to 20+ permissions each. This results in the fact that policies are principally only correctly manageable by (security) experts, except for smaller modifications. It is however desirable that policies can be edited by *domain experts* since they know the – possibly changing – security requirements of their system best. To counter this issue, we propose to express policies in a more abstract way so that security requirements remain clear without getting overloaded by technical details. Our approach to policy modelling uses terms *specific to the application's domain* so that experts of this domain are able to express and understand the security requirements. We demonstrate a possible solution by adopting methodologies from software engineering, especially from model-driven development.

SELinux is an initiative originally initiated by the US National Security Agency developed as an open source product with an active community [2]. It enforces MAC policies based on *Type Enforcement*. In this contribution we show how it can be used for data protection against unauthorised access and how to tackle the aforementioned policy configuration problem for domain experts.

By designing a configuration language specific to a target domain (e.g. healthcare, banking, e-government), we abstract from the underlying technical details. Under the assumption that domain specialists know the security requirements of their systems best, our approach enables them to build enforceable security policies out of which security rules, being enforced by the target system, are generated.

As abstraction necessarily means the omission of details, the creation of templates for policy generation is an important aspect of this contribution. The result enables (e.g. software vendors) to develop large parts of a security policy for their application and to include the possibility of end-user customisation.

The rest of this contribution is organised as follows: first, we shortly describe how MAC frameworks protect sensitive data in Sections 2 and 3. Section 4 presents the components of the model-driven approach and defines a domain-specific language. This language is used and refined in Section 5. We conclude with a discussion on our prototypical implementation and related work (Sections 6 and 7).

2 Background

MAC frameworks for operating systems are gaining growing importance. They are ideal candidates to enforce security requirements at the level of the operating system. However, their enforcement logic resides within the kernel, which means that such systems are mostly dealing with accesses to kernel objects. One reason why policies for such systems are complex is the different semantics of the items to protect and their according kernel objects.

There are different access control frameworks realising MAC, all with similar ambitions, but sometimes varying concepts of realisation (e.g. RSBAC [1] [14], grsecurity[1], SMACK[2] etc.). SELinux, which implements the Flask architecture [24], makes use of LSM hooks [28] and is integrated in the Linux kernel. This implies that some distributions already come with SELinux enabled and a preconfigured policy.

The policy model of SELinux is a combination of *Type Enforcement* (TE) and *Role-Based Access Control*. Access control functionality is split up between different components [12]. The decision whether some subject is granted permission to access an object, takes place in the so-called *Security Server*. Enforcement, however, is done by *Object Managers* (OM). Every object in the system is assigned a *security context* – a label consisting of a *user*, a *role*, a *type* and optionally security level information (`user:role:type`). SELinux further defines a number of object classes. Every object class represents a specific kernel object (e.g. file, process, socket, IPC etc.) and has its own OM. When an object is to be accessed, its OM passes the security labels of the subject and object to the Security Server and waits for a decision. The Security Server reports its decision back to the OM, which enforces it. This decision is mainly based on the type-part of the security context as the Security Server looks up the types of the object and subject (also called *domain*).

There exist other MAC frameworks claiming that their policies are much easier to configure than SELinux policies. In fact, they (e.g. SMACK and TOMOYO[3]) define policies with much less objects and permissions. Hence, policies are smaller and easier to create and understand. However, these advantages also reduce flexibility. Opposed to that, SELinux is covering finer-grained access permissions. Further, it offers the possibility to introduce new object classes and not only enforce access control on kernel objects, but on any kind of logical object. This comprehensive enforcement facilitates access control requirements along each layer in the software stack if OMs are implemented there.

Model-driven techniques enable the consideration of security requirements in early phases of system development and deployment [5]. The overall idea of security (configuration) models is to use them for documentation purposes on one side, but – and this is the point we are focusing on here – also for automated policy generation and feedback channel on errors or exceptional events. Domain-specific modelling techniques render configuration more accessible because it is focusing on the exact core of a specific issue [22].

3 Protecting Data

Comprehensive data protection on a computer system requires a secure underlying operating system (cf. [17]). Before we define domain-specific security policies, we show how SELinux protects sensitive data by restricting access to kernel objects.

[1] Available at `http://www.grsecurity.net`
[2] Available at `http://schaufler-ca.com`
[3] Available at `http://tomoyo.sourceforge.jp`

SELinux already provides the complete architecture needed for enforcing access control. The things left to be done are the configuration of a security policy and the labelling of subjects and objects. The provided Security Server bases its access decision on the label of subjects and objects. TE in SELinux is very flexible [19] and provides transitions among types. An excerpt from such a policy is shown here:

```
domain_type(docViewer_t)
domain_entry_file(docViewer_t, docViewer_exec_t)
#Domain transition to docViewer_t
domain_auto_trans(user_t, docViewer_exec_t, docViewer_t)
allow docViewer_t self:dir {read getattr search};
allow docViewer_t self:file {read getattr write};
libs_search_lib(docViewer_t)
libs_read_lib_files(docViewer_t)
```

The policy shows a few macro calls and two `allow` rules. It is basically responsible for allowing an application to be executed and to transition to its own domain (`docViewer_t`).

For controlling access to certain objects, we assign each subject and object a type and specify which types are granted which permissions on other types. This is done via `allow` rules. In the above example one could for instance define a type `document_t` and allow applications with type `docViewer_t` to `read` such labeled `files`. As one can already imagine, this simple task may become error-prone and cumbersome when permissions to different kinds of objects keep changing constantly.

Another important point is that systems are potentially used by different users. Users usually have different permissions. When a user logs in, it is assigned a security context depending on the system configuration. E.g. user Bob is assigned the security context `bob_u:admin_r:admin_t` whereas user Charlie has the context `charlie_u:clerk_r:clerk_t`.

As both subjects and objects are labelled, it can be decided who may access which files and applications. With *type transitions* it is possible to change the security context of a subject, i.e. a `process`:

```
type_transition clerk_t \
  docViewer_exec_t: process clerkViewer_t
type_transition admin_t \
  docViewer_exec_t: process adminViewer_t
```

The above stated rule specifies[4] that when a user of type `clerk_t` executes a file of type `docViewer_exec_t`, the executing process will transition to `clerkViewer_t` by default (cf. Fig. 1). This allows to specify different permissions depending on who executes an application.

[4] The example is simplified in the sense that it only shows the default transition rule. For such a transition to succeed the source type must have the permission to execute files with the target type, the transition has to be explicitly allowed and the new type must have the `entrypoint` permission to files with the target type.

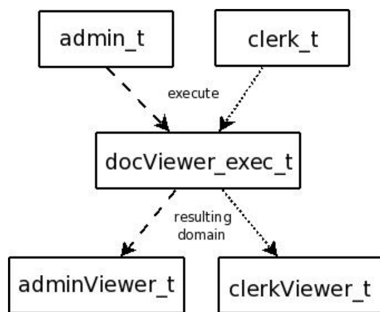

Fig. 1. Executing an application with different users results in different domains

These are the basic building blocks for constructing a comprehensive security policy and its enforcement. There are further statements possible in an SELinux policy which we will not go into detail here. The configuration of such a system is basically about creating types for objects to protect, and granting permissions to subjects on these types. This process is complicated by determining which object classes are appropriate for the items to protect and the large *semantic gap* between them. This semantic gap results from the fact that items one wants to restrict access for by different users, usually have no one-to-one correspondence to kernel objects. However, SELinux protects kernel objects. For example, in the aforementioned policy excerpt consider a change that clerks should be allowed modifying all tax statements and social security documents. For doing such a change one would have to first find out which different `type`s are used for these documents, check to which object classes these `type`s are assigned to, and finally make these changes for all type/object class combinations. Furthermore, even if such a one-to-one correspondence exists for some cases, one still has to "map" logical objects (i.e. the protection items) to kernel objects manually. Hence, a systematic approach is needed to make such a powerful protection mechanism more viable.

3.1 Use Case

We present a use case based on the aforementioned policy statements. This section roughly sketches how an according SELinux policy would look like. At the end of this contribution this use case will be revised using the methodologies and results developed throughout this paper to evaluate its advantage.

Alice is a physician working in a hospital. Every morning before Alice starts her daily work she picks up one of the hospital's mobile PCs. She uses it to enter treatments when medicating patients. The data entered on the PC is synchronised with the hospital database from time to time depending on the current connection status. Continuous connection can not be guaranteed – especially because Alice is also an emergency physician. The data on the mobile PC is synchonised at the latest when it is brought back in the evening. Charlie is the assistant doctor of Alice and is usually always with her. However, he is only

allowed to *view* patient records but not to edit them. Every action on the PCs is logged and the logs are protected against unauthorised modification.

Additionally, the hospital uses the patient records for scientific reasons to evaluate which treatments or therapies are best to treat certain diseases. Of course, these evaluations have to be anonymous so that when collecting statistical data the personal data of patients remains undisclosed to scientific employees.

When additional security requirements have to be enforced in future, e.g. pharmacists should be able to see prescriptions and mark them as "consumed", it is the hospital security administrator's responsibility to make the necessary changes. For this task she will have to deeply look at the current policy to find out what has to be added or changed. A comparable task in software development would be a modification of an application written by another developer without having any documentation at hand describing the application.

4 Model-Driven Security Policy Generation

A direct shortcoming of existing techniques for SELinux policy creation is their low level of abstraction. There is a large semantic gap between e.g. a medical record containing personal information and a `file` inside a `dir`ectory on the `filesystem` containing this information. The policy writer needs to understand the security requirements on the *domain-side* and transcribes them as security rules in the policy. This is a cumbersome and error-prone task demanding experienced users. This task becomes even more complex when taking composite elements into account, i.e. data elements containing other data elements with different security requirements. The patient's personal information is such a contained data element in our use case, which should not be viewable by scientific employees. We have developed a methodology how these policies can be developed in a structured way. In the following, we assume the person creating policies is a domain expert and we explain the identified components.

A *technical implementation* is an IT system supporting business processes of a specific *domain*, e.g. healthcare. Internally, it is dealing with sensitive data objects which here are the subjects of protection for the *policy*. A domain is described by a *domain model*. This is an abstraction of a domain representing the different data objects, how they relate to each other, how they are composed and also in which ways they can be accessed. The elements of the domain model are used to create a *domain security model* (DSM) by a domain expert. The specification of a DSM is manageable by domain specialists because it is a relatively simple language (role-application-permission assignment) for stating security rules, and its subjects and objects are directly stemming from the domain model. Because an enforceable *policy* should be created ultimately, the domain model needs to be enriched by security knowledge. This is done via *templates* which are created by a *security expert*. Templates contain information about how to allow specific actions on data objects. They are always created for a particular technical implementation, i.e. templates for one specific action are

Fig. 2. Components describing the creation of a security policy

different depending on whether a medical record is persisted in a simple file or in a database. The information provided by the DSM and the templates is used by a policy *generator* to produce the final policy. Figure 2 depicts the different connections among these components.

The templates are stored in a template repository which is accessed by the generator when a policy is generated. Each template defines how to allow a specific action, e.g. allow modification of personal data in a medical record. The number of the templates may grow over time while policies for new applications are defined or new permissions are added. However, they remain unchanged when security requirements in the DSM are changed. The connection between templates and DSM elements is established via the domain model. This is explained in more detail in section 5.

The core of our development is the DSM. The domain expert (e.g. domain administrator) is not editing SELinux policies directly where it would be neccessary to know the technical details about which OS- and application-objects are used. Instead, the security policy is developed in a more abstract way and with *domain-based concepts*. Domain specific expressions used in the DSM can be tailored towards any domain where such policies are useful. This practically means that the words and expressions used in the DSM directly stem from the application domain.

Using an abstract security policy language has two major advantages:

1. The domain-specific security policy configuration renders security configuration more accessible to non-framework experts, hence end-user policy adaptation is becoming much more realistic.
2. A methodology to clearly classify domain-specific resources, applications and corresponding permissions is provided, i.e. the domain model. This structured methodology helps defining policy templates. Its creation is much more manageable than creating a whole policy from scratch and the model helps to distinguish responsibilities.

4.1 The Policy Model

Most of the current security models[5] define access restrictions by associating permissions for a subject to an object [21]. Often this assignment is further refined with constraints. As we use SELinux to express our enforcing policies, a subject is represented by a process running on behalf of a certain user. Objects are SELinux specific object classes and they are assigned permissions. This interpretation of role-based access control can be expressed in our DSM. However, in our case the role-permission assignment of RBAC is implemented using type enforcement [4].

Fig. 3 illustrates the abstraction from type enforcement policies and the domain security model. This abstraction reflects the different interpretations of subjects, objects and permissions regarding the different policy specification languages.

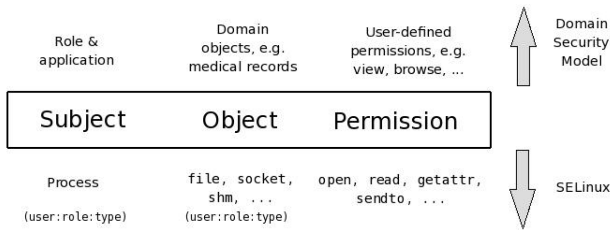

Fig. 3. Abstraction from SELinux policies while still adhering to its policy principles

In a nutshell, the definition of permissions in the DSM results in a number of SELinux permissions. The SELinux permissions are granted on objects of the technical implementation which are represented by the according elements in the domain model. To exemplify this correlation, consider the permission *modify* on medical records mapping to low-level permissions like the action to `open`, `read` and `write` the according `files`. The term *object* from the domain point of view can be mapped to a number of different SELinux object classes. Consider for example an object *hospital network* mapping to a number of internal network nodes within a hospital. SELinux further allows to introduce new objects and permissions. This allows for the enforcement of access control rules also in userland and not for kernel objects only. For *subjects*, a simple one-to-one mapping is not sufficient, because in terms of SELinux, a subject is a process. For DSM-modelling, however, processes are irrelevant. Here one only needs to express which user/role is granted certain permissions. Hence, to establish a connection between users in the DSM and processes in the low-level policy we combine roles and applications. This means that a user executing a specific application[6]

[5] Note that here the term *model* is different from the rest of this paper since it refers to the approach for an access control system and not a system description like a UML model.

[6] Regarding the operating system this *is* a process and a nice mapping for this circumstance can be found for SELinux. We omit the details here.

is granted permissions on certain objects. This completes the basic building blocks of the DSM we are stacking upon SELinux policies. Section 5 will further elaborate on the area of conflict between these two levels.

In short, every model element of our resulting DSM, i.e. roles, application resources and permissions, abstracts from the low-level SELinux policy statements and hides them. This way, the policy remains independent of the underlying SELinux policy statements and the enforcement capabilities of the underlying architecture, e.g. if Userspace Object Managers like SEPostgreSQL[7] or XACE [27] are used. Moreover, it introduces its own terminology since the naming of model-elements is left to the designer of the domain model (see Section 4.2). The template dictates the exact low-level policy statements and is thus responsible for how the DSM statements are really enforced. The same DSM may result in different SELinux policies on different systems if the templates are different.

4.2 The Domain-Specific Model

For a comprehensive definition of a domain-specific language (DSL) [25] we define a metamodel describing the different model elements and how they can be used. In the present case, the domain model consists of resources, applications and permissions on these resources. A *resource* is the abstract form of an object, e.g. `HealthData` (resource) for medical records (object) or `AdminData` (resource) for files containing logs (object). The metamodel further defines the abstract syntax for the DSM. It formalises domain security concepts and policy elements and makes this information accessible to tools so that it becomes machine-processable. The remainder of this section describes the UML profile for (1) the domain model and (2) the DSM.

Defining the DSL: The metamodel basically consists of two different parts which abstract different information: domain abstraction and policy abstraction (see Fig. 4). The *domain abstraction* describes terms and properties of the target domain, e.g. healthcare. This part of the metamodel is used to define the domain model. Conversely, the *policy abstraction* represents how to formulate rules in the DSM, i.e. this is the syntactical part of the developed modelling language. Since a DSM describes elements of a target domain, there exist relationships between the two abstractions.

The Policy Abstraction mostly represents the abstract syntax of the DSM. It defines `Rules`, which associate an `Application` to a number of `Permissions`. Note that unlike RBAC, we are not just assigning permissions to roles directly, since we also want to constrain *how* (i.e. with which application) a certain object is accessed. This necessity is due to the nature of SELinux as described before. To grant a user the permissions of such a rule, its `Role` is assigned to it. This metamodel only supports positive rules and any non-stated permission is denied by default. This is also how SELinux enforces policies.

[7] Available at `http://code.google.com/p/sepgsql/`

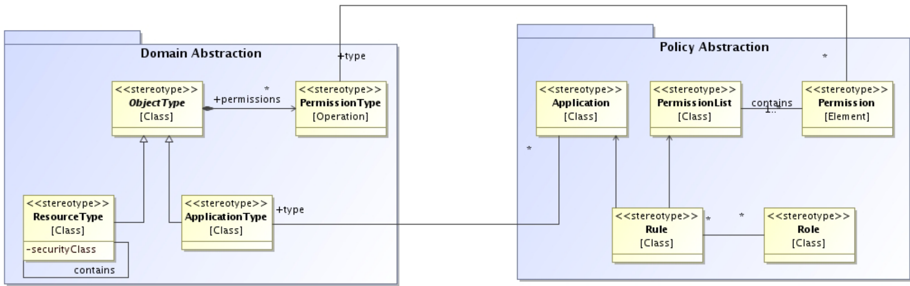

Fig. 4. The security metamodel as UML profile

There are two links which connect the DSM to the domain model: an application in the DSM is of a certain `ApplicationType` and each permission is of a certain `PermissionType`. This way it is possible to use every model element of the domain model in the DSM.

The Domain Abstraction is a formalisation of the different objects and their permissions. `PermissionType`s are part of a general `ObjectType`. This indicates that certain operations on objects need permissions, e.g. view a medical record. There are two different kinds of `ObjectType`s: `ResourceType`s and `ApplicationType`s. The former represents data (e.g. medical record, calendar entry, logfiles etc.), whereas the latter represents applications (e.g. wordprocessor, email client etc.). One crucial property of `ResourceType` is that it defines a containment relation to itself. This indicates that a resource may be subdivided into an arbitrary number of subresources, each with its own permissions. Further, resource types have a tagged value `securityClass`. This is used in the domain model to establish the link between domain model objects and their concrete realisation in the technical implementation, i.e. the SELinux *object class*. The reason why applications are here distiguished from resources is that they possess no containment relation. `ApplicationType`s can however have permissions because applications can for instance communicate with other applications (e.g. via IPC or D-Bus).

Before `ObjectType`s and `Permission`s can be used in a DSM, concrete repesentations need to be created. The result of this is the domain model and as already mentioned this can be done by a domain expert. A similar technique of separating different aspects of a system into distinct models and eventually combine them has already been successfully applied in our previous work (cf. [10] and [11]). This domain-tailoring is done by instantiating metamodel elements, or in our case applying stereotypes accordingly. The hierarchy of objects for a specific domain is composed of a number of elements representing a `ResourceType`.

Fig. 5 shows an example for resource types in the healthcare domain. The most general object is `HealthData`. It defines different specialisations of health data. The semantics of the inheritance relationship between object types is the same as in object oriented design, i.e. derived object types preserve polymorphism. This means that if a certain subject is granted a permission on a `HealthData`

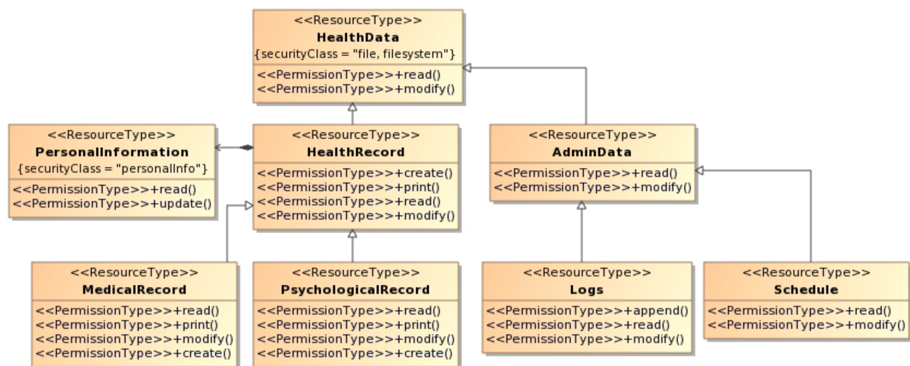

Fig. 5. `ResourceType`s provide a containment and inheritance relation. `Permissions` are directly associated to a resource type.

resource, the same subject is also granted this permission on any resource *derived* from `HealthData` (e.g. `Logs`, `HealthRecord`, `PsychologicalRecord`, etc.) although its concrete manifestation may be different for a subresource if a different template is used to generate the SELinux rules.

The functionality of inheriting granted permissions among objects is not integrated as-is in SELinux. It is however possible to assign attributes to SELinux types and emulate such behaviour. The following listing defines the types for some of the resource types of Fig. 5 and grants processes with type `someApp_t` to `read files` inheriting from resource type `adminData` only.

```
# Type and attribute definitions
type adminData_t, adminData, healthData;
type logs_t, adminData, healthData;
type schedule_t, adminData, healthData;
type medicalRecord_t, healthRecord, healthData;
...
# Rule
allow someApp_t adminData : file read;
```

Note that the previous listing and the model depicted in Fig. 5 are descriptions on different abstraction levels. The model is one artefact which is used to generate such enforceable policies.

Fig. 5 shows that a `HealthRecord` has a special part with additional access restrictions – the resource type `PersonalInformation`. The `PermissionTypes` of contained `ResourceTypes` do not depend on their containers.

Models for `ApplicationTypes` are very similar. As there can also be a hierarchy (inheritance relation) of applications, the DSM allows for specifying rules like "with all `Applications` of a certain kind, a user is allowed to `modify MedicalRecords`."

5 Policy and Template Creation

The missing piece for completing the big picture of model-driven SELinux policy creation are the templates. The basis for their creation is the existence of a domain model with associated security classes (cf. Fig. 6). Templates are created by a security engineer and knowledge about low-level enforcement is neccessary. In this step, one defines the mapping from `PermissionType`s to enforceable SELinux rules. Each `PermissionType` is inspected separately and for each security class of the according object type its enforcement rules are defined. The security engineer sees which items are still unhandled since the object- and permission types are denumerable. In the end, each `PermissionType` is assigned a template so that the generator is able to create the policy. It is furthermore possible to integrate other statements about the source or target types into a permission template, similar to m4 macros in the current SELinux Reference Policy. This aspect is out of scope of the present contribution.

Conceptually, this means for each `ObjectType` the security engineer provides the semantics of any permission. Practically, this is done by assigning `allow`-rules to every security class/permission combination for each `ObjectType`. Table 1 demonstrates how these assignments could look like for `HealthData` in Fig. 5. The example assumes that objects labeled as type `HealthData` are either files or filesystems.

The security engineer iterates over the whole object type hierarchy and defines how every permission should be enforced for every object class. As already mentioned, permissions are inherited by super-object types, and their realisation can be further refined in derived object types.

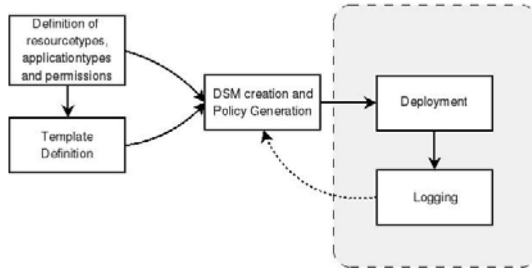

Fig. 6. The whole lifecycle from modelling to deployment. Denials in logs possibly trigger DSM changes by an administrator.

Table 1. Exampe SELinux permission assignments for `HealthData`

	file	filesystem
read	getattr read link	mount remount getattr quotaget
modify	getattr append unlink write create	mount remount getattr associate

The present approach of template creation allows for a clear separation of the tasks where SELinux knowledge is necessary and where it is not. This is already an improvement compared to direct policy editing. However, an even more automatable technique still remains to be found. One possibility could be to use tools like `audit2allow` – which generates `allow` rules out ouf denial logs – and create templates out of its output. This is left for future work.

5.1 Policy Creation

After the templates creation all prerequisites for policy generation are met. The DSM – created by a domain expert – can be realised in different ways: either as a graphical or a textual representation based on the same abstract syntax (cf. section 4.2). Fig. 7 depicts a graphical DSM based on the UML profile developed in this work for the use case example of Section 3.1.

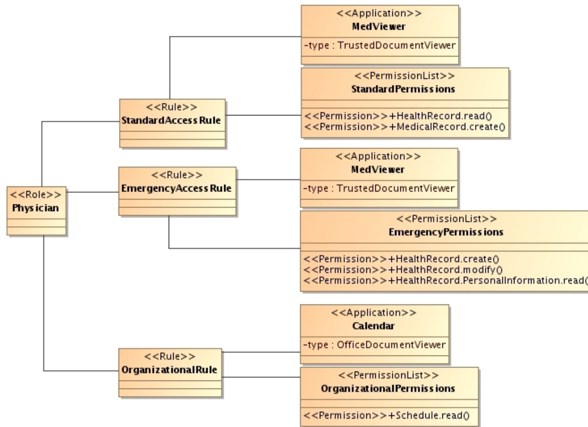

Fig. 7. Example security policy with three rules assigned to a physician

For generating a policy, the generator fetches the corresponding templates and assembles them to the policy implementing the requirements of the DSM. After the generation process, the policy is ready to be deployed on the target system.

When certain actions are denied, the deployed policies produce logs/events. These logs are collected and may be re-used by the security engineer to re-check the DSM or templates. Extracting valuable information out of the logs and use it either for enhancing the DSM or for auditing purposes is subject of future work. The basic idea is to use the model not only as a means of configuration but also as a feedback channel to highlight possible errors or at least the reason for some denials.

6 Implementation

The prototypical implementation of this work consists of two parts: a DSM editor with functionality to generate policies (cf. Fig. 8) and the enforcement target architecture. The policy editor was built using the generator framework openArchitectureWare[8] (oAW). Our predefined security metamodel has been used to generate the textual editor and concrete syntax. For the sake of simplicity, templates are defined in oAW's own template language and no guided template creation functionality was built. To create a policy with our editor, one has to first import or create all necessary policy templates. The generated output is a loadable policy module which can be deployed on systems where SELinux is enabled. If the objects are not yet carrying the right security label, they need to be (re-)labeled with tools provided by SELinux.

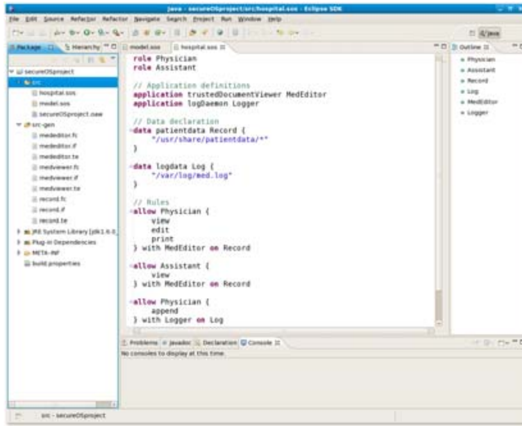

Fig. 8. Domain security model editor showing a textual security policy for hospitals

Enforcement logic is not necessarily residing inside the kernel only. In our use case we make use of the possibility to extend SELinux with additional object classes (`personalInfo`). Object managers for such classes usually run in userland. SELinux already provides means to easily query access decisions from the Security Server. In our example, the application accessing `PersonalInformation`-objects has to enforce the according decisions. We claim that this layer separation has to be taken into account to comprehensively enforce security requirements. Each layer accomplishes a specific task and hence has a specific view on data (cf. [7]).

The enforcement of security requirements inside a userspace application implies modifications to that application. To simplify the modifications on the userspace application we defined some helper methods to keep a clean separation between access control functionality and business logic. A simple example for the method adding prescriptions to a medical record is shown here:

[8] Available at `http://www.openarchitectureware.org`

```
def addprescription(self, prescription):
  if PermissionChecker.checkPermission( \
    prescription.securityLabel, \
    prescription.securityClass ,"create"):
      # do normal work of function...
```

We are currently extending our approach to check such permissions via aspect-oriented programming [8]. This keeps the security-relevant code separated from business functionality.

Another important point is that despite the fact that our prototype only uses two enforcement points, the approach is not limited to that. An existing application making use of this functionality is XACE which confines actions of the XServer. This further expands the possibilities of technically enforceable security requirements. Concretely, this allows to create policies so that no screenshots can be taken from windows showing patient records. All these enforcement rules can be subsumed under high-level security requirements stated in the DSM.

7 Discussion and Related Work

This contribution tackled the problem of complex SELinux policy configuration. First, we identified that the challenge is the *semantic gap* between the policy and the domain objects. Then the different components for making the task more manageable via a model-driven approach were developed, together with a UML profile for expressing the *domain model* and the *domain security model*. An approach for developing the necessary policy templates has been proposed which is based upon the domain model. The work presented a structurised way of policy generation and a clear separation of responsibilities among a security specialist and a domain expert.

A use case from the healthcare domain has been presented and the developed approach has been applied to it. Assuming the necessary policy templates are available, this model can be used to generate an enforceable SELinux policy.

The most error-prone task in this process is still the construction of adequate policy templates. This issue was tackled by structurising it. This makes the process more amenable to tool support, however, understanding of the underlying technology is needed for this task. We think this is an important improvement compared to current low-level policy configuration as it avoids the need to directly edit SELinux policies. Nevertheless, this is an interesting direction for future research.

A question left open in this contribution is how to check the quality and correctness of a generated policy. There is already existing work on how to analyse a generated policy (e.g. [13] and [9]). We are also currently working on our own analysis and consistency-checking method since the existing work mostly takes low-level information flow policies into account and neglects logical objects, which is one of the key benefits in our configuration approach.

Since the concrete SELinux policy is generated, our policy creation method is very flexible and demands no user experience with SELinux by the person

creating the security model, which was one of the stated goals. One of the most promising areas of application is the end-user (i.e. domain expert) adaptation of security policies.

A related approach for the user-guided and semi-automated policy generation of SELinux policies can be found in [18] and [23]. The Polgen-tool processes traces of the dynamic application behaviour and observes information flow patterns. It has a pattern-recognition module and creates new types according the patterns it detects. The policy generation process is interactive and human-guided. In contrast to this we do not aim at creating SELinux policies by observing dynamic application behaviour. Our efforts are towards building a policy generation framework for a specific domain, based on abstract domain-level security concerns and which is easily adaptable by domain specialists. The tools mentioned could however be of great use for creating templates. Another helpful tool in this regard is `audit2allow` which generates `allow` rules out of previously logged denials.

Another related approach can be found in Tresys Cross-domain Solutions (CDS) [26]. To the best of our knowledge it is the only approach apart from ours using graphical models for policy specification. The CDS Framework Toolkit is focusing on cross-domain solutions and not on protecting sensitive data. It deals exclusively with information flow between different applications and networks. The CDS Framework remains very technical although it is using graphical models. Our approach strives to abstract from technical policy configuration and makes it accessible to domain experts. This directly addresses the much criticised configuration complexity of SELinux and renders this issue manageable by end-users. To the best of our knowledge there is no other work aiming at the connection of SELinux objects and logical items to protect for comprehensive access control.

In [16] and [5], the authors introduce the concept of Model Driven Security for a software development process that supports the integration of security requirements into system models. The models form the input for the generation of security infrastructures. However, the approach focuses exclusively on access control in the context of application logic and targets object oriented platforms (.Net and J2EE). Although the initial motivations – namely the "bringing together" of system models with security models – are similar our approach differs in the sense that we are not generating a security infrastructure, but generating policies. Furthermore, we also tackle the problem of template generation which is often not referred by other authors. Another difference to our work is that we target security requirements at various layers of the software stack, the application layer being one of them.

This contribution is a consequent advancement of our previous work on model-driven security. The SECTET Framework provides a target architecture and a model-driven configuration methodology for securing inter-organisational workflows [6] [10]. However, the security infrastructure is located at the application level only in that case. The present work is able to secure the peers in this interaction in a comprehensive way to expand the security infrastructure up to the endpoints [3] and set up a Trusted Computing Base.

References

1. Rsbac - rule set based access control, `http://www.rsbac.org` (last visited April 2009)
2. Security-enhanced linux (selinux), `http://www.nsa.gov/selinux/`
3. Agreiter, B., Alam, M., Hafner, M., Seifert, J.-P., Zhang, X.: Model Driven Configuration of Secure Operating Systems for Mobile Applications in Healthcare. In: MOTHIS 2007 (2007)
4. Badger, L., Sterne, D.F., Sherman, D.L., Walker, K.M., Haghighat, S.A.: Practical Domain and Type Enforcement for UNIX. In: IEEE Symposium On Security And Privacy, p. 66 (1995)
5. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Transactions on Software Engineering and Methodology (TOSEM) 15(1), 39–91 (2006)
6. Breu, R., Hafner, M., Weber, B., Novak, A.: Model Driven Security for Inter-organizational Workflows in e-Government. In: Government: Towards Electronic Democracy: International Conference, TCGOV 2005, proceedings, Bolzano, Italy, March 2-4 (2005)
7. Day, J.D., Zimmermann, H.: The OSI reference model. Proceedings of the IEEE 71(12), 1334–1340 (1983)
8. De Win, B.: Engineering application-level security through aspect-oriented software development. PhD thesis, Katholieke Universiteit Leuven (2004)
9. Guttman, J.D.: Verifying information flow goals in security-enhanced Linux. Journal of Computer Security 13(1), 115–134 (2005)
10. Hafner, M., Breu, R., Agreiter, B., Nowak, A.: Sectet: an extensible framework for the realization of secure inter-organizational workflows. Internet Research 16(5), 491–506 (2006)
11. Hafner, M., Memon, M., Alam, M.: Modeling and Enforcing Advanced Access Control Policies in Healthcare Systems with SECTET. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 132–144. Springer, Heidelberg (2008)
12. ISO/IEC (ed.): ISO/IEC 10181-3:1996 Information technology Open Systems Interconnection Security frameworks for open systems: Access control framework. ISO/IEC, Geneva, int. standard edn. (1996)
13. Jaeger, T., Sailer, R., Zhang, X.: Analyzing integrity protection in the SELinux example policy. In: Proceedings of the 12th conference on USENIX Security Symposium, vol. 12, p. 5. USENIX Association Berkeley, CA (2003)
14. Jawurek, M.: RSBAC-a framework for enhanced Linux system security
15. Latham, D.C.: Department of Defense Trusted Computer System Evaluation Criteria. Department of Defense (1986)
16. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-based Modeling Language for Model-Driven Security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)
17. Loscocco, P., Smalley, S.: Meeting Critical Security Objectives with Security-Enhanced Linux. In: Proceedings of the 2001 Ottawa Linux Symposium, pp. 115–134 (2001)
18. MacMillan, K.: Madison: A new approach to automated policy generation (March 2007)
19. Mayer, F., MacMillan, K., Caplan, D.: SELinux by Example: Using Security Enhanced Linux. Prentice Hall, Englewood Cliffs (2006)

20. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. IEEE, Proceedings 63, 1278–1308 (1975)
21. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-Based Access Control Models. Computer, 38–47 (1996)
22. Selic, B.: A systematic approach to domain-specific language design using UML. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC 2007, pp. 2–9 (2007)
23. Sniffen, B.T., Harris, D.R., Ramsdell, J.D.: Guided policy generation for application authors (February 2006)
24. Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., Lepreau, J.: The flask security architecture: system support for diverse security policies. In: Proceedings of the 8th conference on USENIX Security Symposium, table of contents, vol. 8, p. 11 (1999)
25. Stahl, T., Völter, M.: Modellgetriebene Softwareentwicklung Techniken, Engineering, Management. dpunkt-Verl (2007)
26. Tresys Technology. Cds framework (last visited, April 2009), `http://oss.tresys.com/projects/cdsframework`
27. Walsh, E.: Application of the Flask Architecture to the X Window System Server. In: SELinux Symposium (2007)
28. Wright, C., Cowan, C., Morris, J., Smalley, S., Kroah-Hartman, G.: Linux security modules: general security support for the linux kernel. Foundations of Intrusion Tolerant Systems, 2003 (Organically Assured and Survivable Information Systems), pp. 213–226 (2003)