Reflective Authorization Systems

Massimo Ancona† **Walter Cazzola**‡ **Eduardo B. Fernandez**⋆

† DISI-University of Genova, Genova, Italy
e-mail: ancona@disi.unige.it

‡ DSI-University of Milano, Milano, Italy
e-mail: cazzola@dsi.unimi.it

⋆ Department of CSE-Florida Atlantic University,
Boca Raton, Florida, USA
e-mail: ed@cse.fau.edu

Abstract

A reflective approach for modeling and implementing authorization systems is presented. The advantages of the combined use of computational reflection and authorization mechanisms are discussed, and three reflective architectures are examined for pointing out the corresponding merits and defects.
Keywords: Authorization, Object-Orientation, Distributed Objects, Reflection, Security.

1 Introduction

Security implies not only protection from external intrusions but also controlling the actions of internal executing entities and the operations of the whole software system. In this case, the interleaving between operations and data secrecy may become very complicated and often intractable. For this reason security must be specified and designed in a system from its early design steps [9].
From another point of view

- it is very important that the security mechanisms of the application be correct and stable;
- the security code should not be mixed with the application code, otherwise it is very hard to reuse well-proven implementations of the security model.

If this is not done, when a new secure application is developed the programmer wastes time to re-implement and to test the security modules of the application. Moreover, security is related to: "who is allowed to do what, where and when"; so security is not functionally part of the solution of the application problem, but an added feature defining constraints on object interactions. From this last remark we can think of security as features at a different level and we can separate its implementation from the application implementation.
In our opinion it is possible to use the *separation of concerns* and *transparency*, typical reflection features, to split a secure system into two levels: in the first one there are (distributed) objects cooperating to solve the system application; in the second level, rights and authorizations for such entities are identified, specified and mapped into reflective entities which transparently monitor such objects and authorize the allowed access to the other objects, services or information.
Working in this way it is possible to develop stable and reliable entities for handling security. It is also possible to reuse them during system development, thus reducing development time and costs, and increasing application level assurance.
In most systems authorization is defined with respect to persistent data and enforced by the DBMS and/or operating system. Object-oriented systems define everything as an object, some persistent some temporary, where this separation is not visible at the application level. In these systems authorization must be defined at the application level to take advantage of the semantic restrictions of the information [8]. An early system (not object-oriented) (see [9], page 195), attempted this kind of control by defining programs that had predefined and preauthorized accesses. Reflection appears as a good possibility for this type of control because it does not separate persistent from temporary entities. The Birlix operating system [16] used reflection to adapt its nonfunctional properties (including security) to different execution and application environments.
In this paper we examine how to use a reflective architecture, such as those described above, to manage the authorization aspects of an application and the advantages and drawbacks of using such an approach.

2 Background on Reflection and Security

Computational reflection or just reflection is defined as the activity performed by an agent when doing computations about itself [13]. Behavioral and structural reflection are special cases which involve, respectively, agent computation and structure (for more details see [5]).

A reflective system is logically structured in two or more levels, constituting a *reflective tower*. Entities working in the base level, called base-entities or reflective entities, define the system basic behavior. Entities in the other levels (meta-levels), called meta-entities, perform the reflective actions and define further characteristics beyond the application dependent system behavior.

Each level is causally connected to adjacent levels, i.e., entities belonging in a level maintain data structures representing (or, in reflection parlance, *reifying*) the states and the structures of the entities in the level below. Any change in the state or structure of an entity is reflected in the data structures reifying it, and any modification to such data structures affects the entity's state, structure and behavior.

Computational reflection allows properties and functionalities to be added to the application system in a manner that is transparent to the system itself (separation of concerns) [18]. To this respect, it is useful to consider also *reflection granularity* [1], that is, the minimal entity in a software system for which a reflective model defines a different meta-behavior. A finer granularity allows more flexibility and modularity in the software system at the cost of meta-entity proliferation.

The reflective models considered here are: *meta-object* [13], *message reification* [6] and *channel reification* [2].

In the meta-object model, meta-entities (called *meta-objects*) are objects, instances of a proper class. Each base-entity, called also *referent*, can be bound to a meta-object. Such a meta-object supervises the work of the linked referent.

In the message reification model, the message passing (method call in object-oriented parlance) between two objects is reified by a special object, called *message*. Such a message performs only behavioral reflection.

In the channel reification model, one or more objects, called *channels* are established between two interacting objects. Each channel is characterized by a *kind* which specifies the behavior for message handling by the channel.

We show our ideas using the following scenario: the system is composed of several objects interacting in a client-server manner. Such objects are classifiable, using security concepts, into two non disjoint sets *objects* and *subjects*; where a subject represents an entity performing or requesting an activity (i. e. an active object playing the client role), while an object is a passive entity supplying a service (i. e. a passive object or an active object playing the role of server).

For security reasons the services supplied by a server and the data contained in a passive object are protected and prohibited to some subjects. One way to model such authorizations is the access matrix model (see [12]). In this model the authorization rules are described by a bidimensional matrix indexed on subject and objects and the access to an object o is allowed to a subject s when the item $\langle \mathsf{s},\mathsf{o} \rangle$ of the matrix contains the permitted access type. Such a model can be realized by *capability lists*, *access control lists*, or combinations of these.

A Role-Based Access Control (RBAC) model is particularly suitable for object-oriented systems [15], because accesses can be defined as operations defined in the objects according to the need-to-know of the roles [7]. Because reflection supports a fine granularity of access, its combination with RBAC can be quite effective.

Formally, an authorization right R for a subject s to access an object o, through message (method) m can be written $\mathsf{R(s)=(m,o)}$. Here m usually represents a high-level access type, e.g. hire an employee.

3 How We Model Security With Reflection

A computational system answers questions and supports actions in some applicative *domain*. Following P. Maes [13] we can say that:

> "A system is *causally connected* with its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other."

For example a *control software* is causally connected with the controlled process if changes of the state of the physical process are reflected by changes of the control software state and vice-versa. To this purpose a control software system incorporates structures representing the state of the controlled process. If we consider authorization rights added to a critical control system we note that the causal connection of authorization with the application domain is *indirect*. Authorization (and more generally security) defines capabilities related to human and/or mechanical agents concerned with the responsibility of performing tasks that are considered critical for the controlled system behavior. Thus, an authorization validation mechanism is a *second level* feature causally connected with software agents and objects performing control tasks. This is our motivation for adopting a reflective architecture for implementing authorization schemes. In other words, security specifications correspond to nonfunctional specifications [7] and should be implemented into a meta-level.

Using computational reflection we can separate the authorization control mechanism from the application. Moreover, the meta-level should be protected from malicious intrusions and attacks from the base-level or from other processes. A solution could consist in executing the meta-level in a different address space as an independent process, communicating with the base-

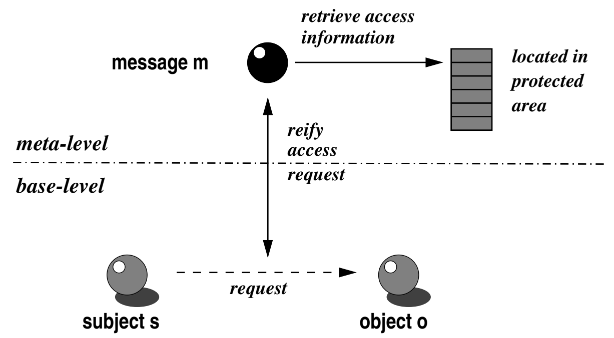

Figure 1: **Message Reification Model to Model Security**

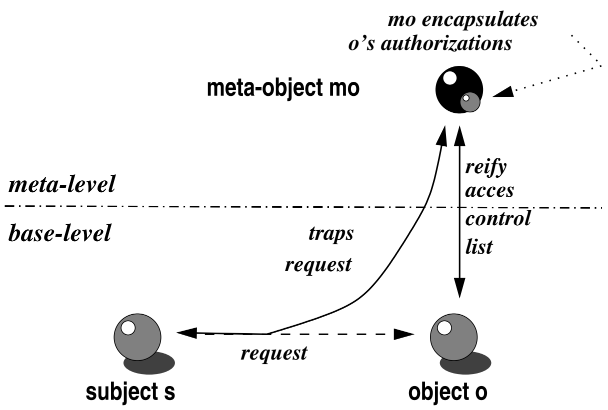

Figure 2: **Meta-Object Model to Model Security**

level via a mechanism similar to the *local procedure calls* (LPC) of Windows NT [4]. LPC is a locally optimized form of the well known mechanism of *remote procedure call* (RPC) of Unix and other systems: LPC is a message-passing mechanism through which clients make requests to servers and used for the server's reply.

In order to avoid confusion with objects in object-oriented systems, we use the term *base-objects* to refer either to subjects and objects of authorization terminology. Moreover, we call *service request* (or method call) every access performed by a subject on an object. There are three possible reflective architectures suitable for controlling authorization rules: the *message reification model*, the *meta-object model* and the *channel reification model*. Each of them provides features suitable to handle different security aspects.

In the message reification model each request m delivered by a client s to a server o is reified into an instance of message m. This object retrieves the related role right R(s) = (m,o) from a predefined and protected area (for example, encoded into data structures of the meta-level); then, it either validates the request by delivering the message to its destination o, or invalidates the request by raising an access violation exception. The validation phase is hidden to normal computation of the system (see Figure 1). The message reification model is suitable for implementing *uniform* authorization mechanisms, not requiring a variety of specialized mechanisms. This model supports specialization only for kind of message, not for combined client and/or server and message kind. Role rights cannot be encoded into the reified meta-entity because if its short life-cycle. Addition and deletion of authorizations must be managed with a different mechanism.

In the meta-object model a meta-object is associated with each object (as shown in Figure 2); this meta-object reifies the related access control list of the referred object. Each request performed by a subject s to an object o is trapped by the related meta-object mo, which evaluates the authorization request. Then, it either rejects the request by rising an exception, or it delivers the request to the original destination s. This model is suitable for implementing highly specialized role rights (specialized per object, per subject or per service request).

Also, each meta-object may hold the methods necessary to modify authorizations rules of the base-objects [10], which can be used to extend or reduce the existing authorizations. In this way, the programmer of the base-objects does not worry about the validity of authorizations changes, which is encapsulated into the meta-object behavior.

A reflective mechanism is able to model more complex authorization schemes than the ones shown above. For example, the authorization rules can be refined by the concept of access mode (such as the read, write and execute access of the Unix file system). This situation corresponds to modifying the value of the elements of the access matrix from boolean to multi-valued data (eg. the symbols *r, w* and *x*). Obviously the presence of more access modes complicates the authorization validation process. In our scenario, we can continue to use the meta-object model and entrusting the entire validation process into the corresponding meta-object, increasing and complicating the meta-object code. Complete workflow authorizations requiring specific sets of rules involving sequences of message sending can be managed with this model.

Another approach is to use a reflective model with a finer granularity than the meta-object one and to entrust each access mode into a different reflective entity. Using the channel reification model [2] we can define a channel kind for each possible access mode. Each element of the access matrix is reified by more channels (one for each value assigned to the matrix item). For example, when the request performed by s to o has write access mode, such a request is trapped by the channel with kind *write* established between s and o which validates it (see Figure 3). In this way the authorization validation process is simplified: each channel checks few authorizations and there exist reflective entities only for base-entities needing an authorization validation protocol. Obviously, the main drawback of this approach with respect to the meta-object approach is the higher number of reflective entities involved.

Security models based on communication flow [3] can be modeled by the channel reification model: in particular, models based on the concept of *flow* like Escort of Scout [17] and

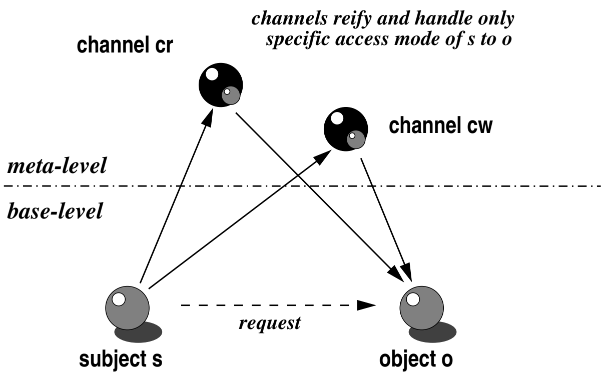

Figure 3: **Channel to Model Access Mode Authorization**

Corps [14]. A path is a first class object encapsulating data flowing through a set of modules. A path is a logical channel made up of data flow connecting several modules. Other two security mechanisms adopted in Escort are *filters* and *protection domains*. A filter restricts the interface between two adjacent modules. However, filters include no mechanism to ensure that a module does not bypass the interface by directly accessing the memory of the other module. A protection domain is a boundary drawn between a pair of modules to ensure that the mutual access is performed only through the defined interface. Filters and protection domains may be modeled respectively by standard reflective channels and *protected* channels, i.e., channels executing in a different address space. The concept of path is more complex and requires an extension of the channel reification mechanism. A channel controlling a path may be obtained by piping or composing the channels controlling the sequence of modules forming a path or by defining a complex channel successively controlling the interface of a sequence of modules in a path. Another approach could be based on a three-level reflective tower, but this approach increases the system complexity without significant advantages.

4 Evaluation

Using computational reflection to include an authorization mechanism into a software system offers many advantages during software development. We can specify, develop, implement and test the modules which implement authorization mechanisms separately from the rest of the application. In this way we encourage reuse and improve software stability. Moreover, the authorization process is hidden to the application entities, thus the code of such entities is simplified. While this can be accomplished by current DBMS authorization systems, we are now controlling access to all executing entities, not just the application's persistent data.
The advantages from the security point of view are that only the entities performing the validation of the authorizations of an object know its authorizations constraints. In this way authorization leaking is minimized.
A first drawback is that flexibility has a cost in terms of efficiency. The Achille's heel of using reflection to realize security could be its implementation mechanism. In fact, the trap actions (shift-up and shift-down) are critical actions. It is possible for a malicious user to intercept the trap and to hijack the request to another reflective entity which will authorize the request. For this reason it is important to protect the meta-level and the access to it. From another point of view this drawback is, also, a strong point of the model with respect to standard authorization models; we minimize the vulnerable points to known system locations that can be more efficiently protected. The possibility of using different address spaces (e.g., different processes) to implement the two reflective layers with kernel system intervention for exchanging information represents an improvement to the security of the complete system.
Some recent proposals to control actions of Java applets have similar purposes to our proposal. In those systems, e.g. in [11], execution domains are created and enforced for specific applications using downloaded content. However, the objective of these approaches is to control access to operating system resources, e.g. files, memory spaces, ..., not to control high-level actions between objects.

5 Conclusions

Reflection offers several advantages when used to model authorization mechanisms. Its main advantage is due to separation of concerns and modularity. Authorization mechanisms can be designed within the application from early development stages, but, at the same time, they can be maintained separate both from the logical and implementative point of view. This fact improves reusability of both functional and authorization software and supports an independent testing of both. More important, it controls access to all executing entities, not just to persistent data.
Another advantage is the ability of implementing a protection layer around the authorization software, thus making the system more robust to unauthorized attempts to change role rights. Obviously, there are also drawbacks: the first is a reduced execution efficiency; flexibility costs in efficiency. The second problem could be represented by the protection mechanism around the authorization layer (meta-level). Running it in a different address space may make programs too inefficient for most applications. Thus, more efficient protection mechanisms, not performing a complete context switching, should be designed. Hardware capability systems appear promising for this purpose. Moreover, the existence of such a protection makes more *understandable*, to malicious users, *where to address attacks to the fortress* and easier to discover Achille's heels.
Future developments are represented by complete workflow authorizations combining specific sequences of service requests: they require that the meta-entities controlling the activated server objects, interact with each other for discriminating le-

gal sequences from illegal ones. More complex authorization schemes may require the introduction of meta-meta-levels, i.e., to raise the reflective tower and the system complexity.
Finally, some prototyping of the proposed architecture and practical experiments will improve the understanding of the role played by reflection in the implementation of authorization systems of high assurance. In particular, its possible use to control the actions of downloaded content would be of high practical interest.

Acknowledgment

We are most grateful to the anonymous referees whose advice advice stimulated further deepening on the topic.

References

[1] M. Ancona, W. Cazzola, G. Dodero, and V. Gianuzzi. Channel Reification: a Reflective Approach to Fault-Tolerant Software Development. In *OOPSLA'95 (poster section)*, page 137, Austin, Texas, USA, on 15th-19th October 1995. ACM. Available via anonymous ftp at `ftp.disi.unige.it/ftp/person/CazzolaW`.

[2] M. Ancona, W. Cazzola, G. Dodero, and V. Gianuzzi. Channel Reification: A Reflective Model for Distributed Computation. In *proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, 98CH36191, pages 32–36, Phoenix, Arizona, USA, on 16th-18th February 1998. IEEE.

[3] W. E. Boebert and R. Y. Kain. A Pratical Alternative to Hierarchical Integrity Policies. In *proceedings of 8th National Computing Security Conference*, October 1985.

[4] H. Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[5] F.-N. Demers and J. Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *proceedings of the workshop section*, in IJCAI'95 (International Join Conference on AI), Montréal, Canada, August 1995.

[6] J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.

[7] E. B. Fernandez and J. C. Hawkins. Determining Role Rights from Use Cases. In *proceedings of the 2nd ACM Workshop on Role Based Access Control (RBAC'97)*, pages 121–125, November 1997.

[8] E. B. Fernandez, M. M. Larrondo-Petrie, and E. Gudes. A Method-Based Authorization Model for Object-Oriented Databases. In *proceedings of the OOPSLA'93 Workshop on Security in Object-Oriented Systems*, pages 70–79. ACM, 1993.

[9] E. B. Fernandez, R. C. Summers, and C. Wood. *Database Security and Integrity*. Addison-Wesley, Reading, Massachusetts, 1981.

[10] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communication of the ACM*, 19(8):461–471, August 1976.

[11] T. Jaeger, N. Islam, R. Anand, A. Prakash, and J. Liedtke. Flexible Control of Downloaded Executable Content. http://www.ibm.com/Java/education/flexcontrol, 1997.

[12] B. W. Lampson. Protection. *Operating System Review*, 8(1):18–34, January 1974. Reprint.

[13] P. Maes. Concepts and Experiments in Computational Reflection. In *proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*, pages 147–156. ACM, October 1987.

[14] E. Menze, F. Reynolds, and F. Travostino. Programming with System Resources in Support of Real-Time Distributed Applications. In *proceedings of the 1996 IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pages 36–45, Laguna Beach, Ca, February 1996. IEEE.

[15] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, February 1996.

[16] S. Sonntag, H. Härtig, O. Kowalski, W. Kühnhauser, and W. Lux. Adaptability Using Reflection. In *proceedings of the 27th Annual Hawaii International Conference on System Sciences*, pages 383–392, 1994.

[17] O. Spatscheck and L. L. Peterson. Escort: A Path-Based OS Security Architecture. Technical Report TR-97-17, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, November 1997.

[18] R. J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.