Hidden Software Capabilities

D. Hagimont, J. Mossière. X. Rousset de Pina, F. Saunier

Laboratoire IMAG-LSR, 2 av. de Vignate, 38610 Gières - France

Internet: Daniel.Hagimont@imag.fr

Abstract: Software capabilities are a very convenient means to protect co-operating applications. They allow access rights to be dynamically exchanged between mutually suspicious interacting applications.

However, in all the proposed approaches, capabilities are made available at the programming language level, requiring application developers to wire protection definition in the application code, which is detrimental to both flexibility and reusability. We believe instead that capabilities should be hidden from the application programmer, allowing protection definition and application code to be clearly separated.

In this paper, we propose a new protection model based on *hidden software capabilities*, in which protection definition is completely disjoined from the application code and described in an extended Interface Definition Language (IDL). This allows to specify protection for existing modules and to easily change the protection policy of an application.

This protection model can be integrated in a wide range of operating systems. We are currently implementing it in a single address space operating system based on distributed shared memory.

1 Introduction

Protection is a crucial aspect of distributed computing, in particular when users co-operate using shared objects or shared memory. Several protection models have been proposed [Organick 72, Wulf 74, Kowalski 90, Cabrera 92, Hagimont 94], among which capability-based protection [Levy 84]. Capabilities are very convenient because of their flexibility, allowing the implementation of various execution and protection models.

However, in all the proposed approaches (e.g. [Tanenbaum 86, Chase 94]), capabilities are made available at the programming language level through capability variables that are used explicitly for:

- accessing objects (mapping an object to a capability)
- changing protection domains (with a capability on the called domain's entry point)
- transferring access rights between protection domains (passing a capability as a parameter).

Therefore, in a protected application, the definition of protection is wired in the application code, which necessitates a different implementation for each distinct set of access rights associated with its execution. Moreover, protecting an already developed application or reusing an unprotected module implies at least a partial re-implementation.

We believe instead that capabilities should be invisible to the programmer, in the sense that it should be possible to define the protection policy of an application independently from the application code. This allows the execution of the same code with different protection policies, which ensures both economy and separation of concerns. We propose a new protection model which is based on capabilities that are hidden from the application programmer, and which allows defining the protection policy of an application only in terms of the application's interface. The idea is implemented by managing capabilities in a separate protected layer (the operating system kernel or one of its extensions), and by providing administrators with an extended interface definition language (IDL [OMG 91]) in which the management of access rights can be expressed and from which adequate "protection stubs" can be generated. This results in a system in which application code does not depend on protection; protection-related actions (access rights checking, domain-crossing and rights transfers) are only triggered when the application traps into the protected layer.

In this paper, we present this new protection model and its integration in the Arias Distributed Shared Memory (DSM) system which is currently under developement in the IMAG-LSR laboratory. However, we believe this protection model can be applied to many other systems and we are investigating its integration in the CORBA architecture.

The paper is structured as follows. In section 2, we provide an overview of capability-based protection and we motivate the introduction of hidden capabilities by pinpointing the limitations of previous work. Then, we explain in section 3 how an IDL-based system can help meeting the requirements of our protection model. Section 4 describes the integration of this model in the Arias DSM systems. We conclude in section 5.

2 Capability-Based Protection

2.1 General Protection Requirements

The purpose of protection is to control the actions of system entities, called agents[1], able to produce or to propagate errors on system controlled entities, called objects. We assume that an object is defined by the operations that access it. In a system which allows objects to be shared between agents, protection mechanisms must control, for each agent in the system, the objects it can access and the operations it can invoke on them. Furthermore, the protection mechanism must be safe and flexible. Safety means ensuring that independent entities are mutually isolated. Flexibility means that different access rights may be provided for different operations on an object, and that access rights may be extended or restricted depending on both the agent and the current context. Last, but not least, the protection mechanisms must cater for mutually suspicious applications. Two categories of models have been proposed in order to achieve these goals; those based on access lists and separated execution contexts, and those based on capabilities and protection domains. The latter model is usually regarded as the more flexible; we examine it in the next section.

[1] Usually implemented by processes or threads.

2.2 The Capability Model

The capability model is based on the concepts of capability, protection domain and inter-domain invocation capability.

2.2.1 Capabilities

A capability is a token that identifies an object and contains access rights that define the allowed operations on that object. In order to access an object, an agent must own a capability on that object. When an object is created, a capability is returned to the creator and usually contains full rights on the object. The capability can thus be used to access the object, but can also be copied and passed to another agent, providing it with access rights on that object. When a capability is copied, the rights associated with the copy can be restricted, in order to limit the rights given to the receiving agent.

2.2.2 Domains

Domains are protection environments that are defined in order to allow agents' rights to evolve in a controlled fashion. A *domain* is characterized by the set of capabilities it contains, which defines an access window on the global object space, including other domains. At a given time, an agent executes in exactly one domain and can only perform those operations authorized by the capabilities included in that domain. An application may be composed of several domains. Thus, domains provide isolation between independent entities within an application.

An agent may move to a different protection domain in order to change its rights according to the current needs, by performing a cross-domain invocation. In order to control this invocation, each domain *D* exports an interface, which specifies the subset of its operations that can be invoked from other domains. Each of those operations is called an entry-point of domain *D*, and is actually implemented by a procedure for which *D* has a capability. The invocation of an entry-point of a domain *D* is a protected operation, which is permitted only from those domains which include a special capability on *D*, called an inter-domain invocation capability, or for short a *domain capability*. In the following section, we analyze the rights that a domain capability must include.

2.2.3 Domain Capabilities

A call on a domain entry-point may include parameters; therefore, the invoked operation must receive enough rights on the parameters to be able to execute successfully. We assume that the domain capability not only permits the invocation of its target domain-entry point, but also specifies the rights which have to be transferred from the current domain to the target domain and conversely.

The rights needed depend on the specification of the entry-point parameters, which may include data and procedures.

- *Data*. Data may be passed by value or by reference. In the first case, the data is copied in the invoked domain and no rights need to be passed. But, in the second case, the caller domain must give rights on the referenced data. These rights must

be greater or equal than those needed by the invoked entry-point to process the data. Generally, if the principle of mutual suspicion is respected, the rights provided by the caller are exactly the rights requested by the callee. Furthermore, if the data is a result parameter, the caller domain must accept the installation of a new capability either within itself (if it trusts the computation carried out by the callee domain) or in an other domain it specifies.

- *Procedure (or method).* As for data, there are two ways of passing a procedure as a parameter: by value (i.e. copying executable code), or by reference (i.e. passing a pointer to the procedure). But the risk of having the procedure used as a Trojan horse would compel most of the time the callee domain to request the right to invoke the procedure in another domain rather than the right to process it directly. This means that the capability associated with a procedure parameter must be a domain capability rather than a simple execute capability. In turn, this domain capability must specify all the rights needed by the procedure to execute successfully. For the reasons already given, if the procedure parameter is a result parameter, the caller domain will only give the right to install a domain capability in itself when the operation returns.

For input parameters, the callee domain specifies the rights it needs and the caller domain the rights it offers; conversely, for output parameters, the callee domain specifies the rights it offers and the caller domain the rights it accepts. The construction of a domain capability and its installation in a domain can be done only when the offered rights include the needed rights. The rights transferred must be identical to the rights needed. Furthermore the caller should have the possibility to specify the domain in which the capabilities associated with the results have to be installed.

In order to illustrate capability based protection, let us consider the case of a *Print* procedure, exported by a printing server, that allows a client to print out a file (Figure 1). We assume that the client trusts the printing server.

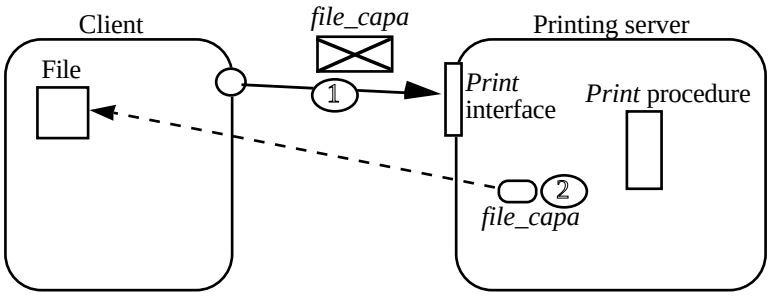

Figure 1. A printing server

The *Print* procedure needs to execute in supervisor mode in order to access the device driver associated with the printer, but this privilege is not granted to the clients. Therefore, the *Print* procedure executes in a separate protection domain, and a domain capability is delivered (when the printer is installed) to the clients, providing them with a means to invoke

the protected procedure. When a client wants to print out a file, the *Print* procedure needs to get read rights on this file; therefore the user will, at invocation time (1), pass a read-only capability on the file (*file_capa*) to the printer protection domain. This capability allows the *Print* procedure to read the contents of the file (2).

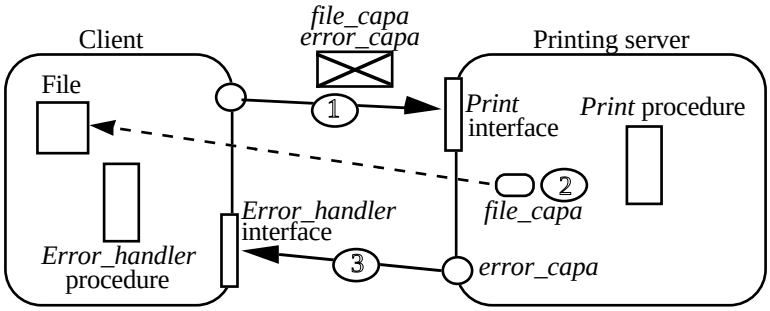

Figure 2. Domain capability parameter passing

Usually, the printing of the file executes asynchronously, the *Print* procedure returning before the completion of the printing. In order to illustrate capability parameter passing, let us assume that the *Print* procedure allows the client to pass an error handling procedure to the printing server, to be called in case the printing fails (Figure 2). Since the printing server does not trust the client, this procedure must be executed in the client protection domain. Therefore, when the client invokes the *Print* procedure, a domain capability (*error_capa*) is passed (1) to the server. If the printing fails, the *error_capa* capability is used to invoke the *Error_handler* procedure in the client protection domain (3).

2.3 Implementations of Capabilities

Capability-based systems have evolved from hardware capabilities to software capabilities. We briefly recall the principles of both implementations, and we then present the requirements we place on our capability-based protection model.

Hardware Capabilities

Early attempts to manage protection by means of capabilities were based on specific hardware. Several capability-based hardware addressing systems [England 75], [Wilkes 79] were built for the management of protected shared objects.

These systems are characterized by the way objects are addressed at run time. The machine has a set of capability registers. Operands in memory are addressed in a machine instruction through a capability register. Therefore, capabilities are both used for addressing and for protecting objects. In order to ensure their protection, capabilities are grouped in hardware-protected segments called C-lists.

These machine and system architectures were very popular in the 70s, but the standardization of the hardware and the widespread use of the Unix system stopped the trend towards capability based architectures.

Software Capabilities

In a second step, software capability based systems were designed on standard hardware, capabilities being protected by encryption. In these systems, the standard addressing mechanism of the underlying hardware (i.e. virtual addresses) is used for addressing objects, and capabilities are only used for object protection and access control.

A first example is the Amoeba system [Tanenbaum 86], based on the client-server paradigm. A server is a protection domain in which objects are accessible using their virtual addresses. An object is managed by exactly one server. A server can create and export domain capabilities to other servers, allowing external processes to enter the server to invoke operations on objects within the server. Interaction between clients and servers is message-based. A process that invokes an operation on an object in a distant server sends a message to the server, passing the capability for the operation. The receiving server checks if the capability is valid and, if so, performs the operation. In order to prevent clients from forging rights on objects, exported capabilities are encrypted, and may only be interpreted by the server. Capabilities can be passed as parameters of method invocations, allowing rights to be transferred between servers.

Another example is the Opal single address space system [Chase 94]. The main difference with Amoeba is that objects in Opal may be shared between several protection domains. A process in one protection domain can either map an object locally using an object capability, or enter another domain using a domain capability called a *portal*.

Software capability based systems are very well suited for the development of protection applications; however, as shown in the following section, current systems lack flexibility.

Hidden Capabilities

In all software capability based systems, capabilities are made available at the programming language level through capability variables that are used explicitly for:

- accessing objects (mapping an object to a capability)
- changing protection domains (with a capability on the called domain's entry point)
- transferring access rights between protection domains (passing a capability as a parameter).

In the Amoeba system, capabilities are used explicitly to invoke objects managed by a server. Capabilities are also explicitly exchanged as parameters when an invocation takes place between two servers. In the Opal system, capabilities are used in the application code to attach (i.e map) segments in protection domains; a process uses portals in the same way to enter a new domain. Therefore, in these systems, the definition of protection is wired in the application code; as a consequence, a different implementation of the same code is needed for each distinct set of access rights associated with it. Moreover, protecting an already

developed application or reusing an unprotected module entails at least a partial re-implementation.

This drawback - managing capabilities in the application code - has already been partially addressed in a few systems. For instance, in Opal, it is possible to force a trap into the system when an object is not yet mapped in the current protection domain, and to have the object automatically mapped if a capability for that object is associated with the protection domain. The advantage of this scheme is that only object names (virtual addresses in Opal) have to be managed at the application level if the object can be mapped locally. Furthermore, Opal allows a portal (i.e. an inter-domain invocation capability) to be hidden from applications behind a proxy object [Shapiro 86]. However, with this approach, the same object cannot, with one protection setting, be called locally and, with another setting, be invoked in another protection domain. Moreover, capability parameters passed to the target protection domain must be specified in the application code.

We believe that it is possible to go one step further and to make application code *completely* independent from capabilities. We believe - and our implementation is described in the following sections - that applications should only see virtual addresses or names; access control is ensured by the system, using capabilities, cross-domain invocations when needed, and cross-domain capability transfers as required.

If the current protection domain contains a capability for the target operation, then the operation is performed locally. If it does not, but the domain contains a domain capability (associated with the target operation), then the same access should trigger a domain crossing, allowing the operation to be executed in the target domain.

One of the main problems is then to allow capability parameters to be exchanged between protection domains while keeping code independent from protection. From now on, we assume that an operation on an object has a procedural form. We show in section 3 that capability transfer rules can be expressed using an Interface Definition Language (IDL), that allows the generation of adequate system code for capability passing and control.

3 Expressing Protection with an IDL

In this section, we present how an interface definition language can be extended in order to specify the protection policy associated with an application.

3.1 Motivations

IDLs are generally associated with Remote Procedure Call (RPC [Birrell 84]). An IDL is used to specify the interface of a service, allowing the stub generation for the marshalling of parameters.

The key feature we are using here is that an interface is defined separately from the code of the application. Just like an interface is associated with an entry point in a remote server, an interface can be associated with a domain capability which is an entry point in the target protection domain. This interface would actually exist anyway if protection was implemented by a client-server system.

Therefore, our proposal is to specify capability transfer rules in an interface definition associated with each domain capability.

3.2 The Protection-enhanced IDL

Let us recall that an entry point of a domain is defined by a procedural interface (see example in section 2.2.3). This interface is associated with the domain capability that gives access to the domain. We propose to use an IDL to define the procedural interface to a domain. This IDL is extended to specify protection rules that define the capabilities which must accompany object names transmitted as parameters. Note that this specification is associated with a domain capability and is completely separated from the application code.

The interface of a domain capability including protection statements is called a Protected Procedure Interface (PPI). We now examine the contents of a PPI.

To meet all its requirements, a PPI should allow:

- the called domain to control the capabilities that enter (at call time) and leave (on return) the domain,
- the caller domain to control the capabilities that leave (at call time) and enter (on return) the domain.

Therefore, to fully control the access rights, both the called and the caller domains must define PPIs that correspond to their respective requirements; then the system has to match these requirements in order to check their compatibility. This matching is done when the domain capability is installed in the caller protection domain; it fails if the PPIs are not compatible.

As stated in section 2.2, the called domain needs to specify the minimal rights it needs for in-parameters and the maximal rights it accepts to give for out-parameters. On the other side, the caller domain specifies the maximal rights for in-parameters and the minimal rights for out-parameters. The result of the matching process is to install in the calling domain a capability specifying the entry point to be called and the rights required on parameters by the caller and the callee. At execution time, the addresses of effective parameters are used to search the two domains for a set of capabilities including the required rights.

For example, let us consider again the printer example described in section 2.2.3. The client protection domain can enter the server protection domain using a domain capability. Then, the PPI associated with this domain capability is the following:

```
operation Print (
        in obj_capa(read) object-type  file,
        in dom_capa(interface) proc-type  error_handling,
);
```

This PPI is the one specified by the server. *Object-type* represents the type of an object name (e.g. a virtual address). The server requires an object capability with at least *read* rights with the parameter *file*. The server also requests a domain capability to be passed with object *error_handling*. For this domain capability, the server is the caller and the client the callee; *interface* is the PPI that expresses the requirements of the client for this domain

capability parameter. When the effective domain capability is passed (for the formal parameter *error_handling*), the system will look for a capability containing a capability compatible with interface.

Separately, the client domain can specify its own PPI (for the *Print* entry point) that will be matched to the server's PPI when the capability is installed in the client protection domain.

3.3 Extensions to the IDL

In order to demonstrate the power of expressing protection rules with an IDL, we now present some possible extensions that we plan to include in our implementation.

3.3.1 Management of Complex Structures

The first extension is related to the management of complex structures. The IDL can be enriched to allow the transfer of a set of capabilities, associated with a complex structure.

With current IDLs, it is easy to specify that a linked list structure has to be passed as a parameter of an invocation. The stub generated from the interface will, at run time, follow pointers and transmit all the objects included in the list, and finally rebuild the list in the remote environment.

We propose to use the same mechanism in order to transfer a set of capabilities associated with a complex structure such as a linked list. Just like an IDL declaration can specify the transfer of a linked list, the following PPI specifies the transfer of the capabilities on the objects contained in a linked list:

```
type list-type =
        record
                obj_capa(read) object-type   file;
                obj_capa(read) list-type   next;
        end;
        operation  Print (
                in obj_capa(read) list-type  list_of_file );
```

In the same way than a stub generated by an RPC compiler does [rpcgen 88], the *list-type* structure will be searched recursively in order to collect the capabilities that must be transferred. In the example, the PPI specifies the transfer of read rights on the objects in the list and on the linkage structures of the list. It is then used to transmit a list of *file* objects to the *Print* operation.

3.3.2 Administration of Protection Domains

In order to provide functions for the administration of capabilities in protection domains, we plan to manage lists of capabilities. A symbolic name is associated with a list of capabilities and a list may be included in a list, allowing the domain administrator to organize access rights in a domain as a hierarchy like directory trees in the Unix system.

Thus, in order to facilitate the administration of capabilities, the IDL can be extended to allow the installation of capabilities in this environment. When a PPI specifies the transfer of a capability to a protection domain, it is then possible to control where the capability should be installed in that protection domain.

In the following example, two operations are defined for the management of a library. When a document is registered in the library, the capability on the document is stored in the *bib_doc_list* list.

```
operation Biblio_Register (
        in obj_capa(rights) doc-type doc install bib_doc_list,
        out String[10] refbib );
```

On the other side, the IDL can also allow the administrator to place restrictions on capabilities leaving the domain. The client domain that accepts the domain capability associated with this PPI can specify the following PPI, meaning that only capabilities from the *export_doc_list* list may leave the client domain.

```
operation Biblio_Register (
        in obj_capa(rights) doc-type doc from export_doc_list,
        out String[10] refbib );
```

The system matches these two PPIs when the domain capability is installed in the client domain.

3.4 Programming with Hidden Capabilities

In this section, we illustrate the use of hidden capabilities and PPIs for the development of a protected application based on existing software components.

Let us consider the case of a desktop application that is composed of several reused modules such as: the library and printer modules mentioned previously, tools for the administration of the local host and others. The Desktop manager uses the interfaces of these modules, but these modules may also reference each other (Figure 3).

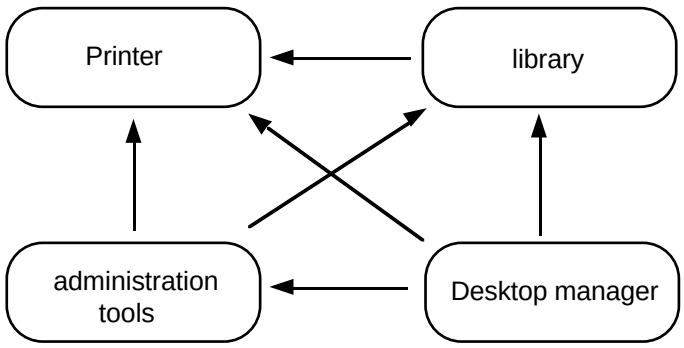

Figure 3. Inter-module references

In this figure, arrows denote module references. For example, the Desktop manager uses procedures exported by the library, the administration tools and the printer modules.

The code of these modules is completely independent from protection specifications. The administrator defines the capabilities associated with the objects managed by the application. These capabilities may allow some of these modules to execute locally, i.e. in the protection domain of the user, but it may also force the execution of some other modules in separate protection domains in order to protect internal data from untrusted users.

The code of the desktop application and the modules will always use objects as if they were available locally; the management of access rights is defined by the installation of capabilities in the involved protection domains and by the specifications of the PPIs.

In order to clarify this, let's give some details about the communication between the desktop and the administration tools (Figure 4).

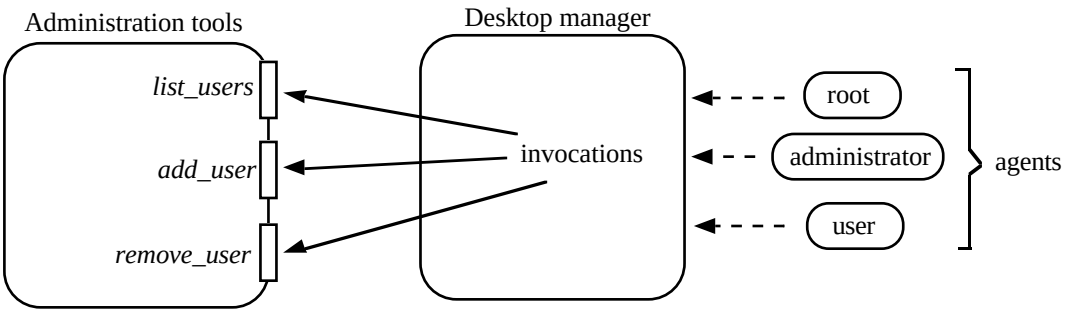

Figure 4. Protection independent invocation code

When the Desktop manager is used in *root* mode, it is executed in a domain containing object capabilities for the three entry points (*list_users*, *add_user*, *remove_user*) that will be invoked in that domain (assuming the *root* user knows what he is doing). When the Desktop manager is used by a *system administrator*, then it is activated in a domain containing domain capabilities for the three entry points, allowing administration tools to be executed in a secure environment. When the Desktop manager is used by a regular user, only a domain capability for the *list_users* entry point will be provided.

We conclude this section by pointing out a more intuitive advantage of hidden capabilities. The separation of the implementation code from the protection policy definition makes the application more easy to maintain.

4 Integration in an Operating System

In this section, we describe the integration of our protection model within an operating system: the Arias single address space distributed shared memory system, which is currently under development.

We first describe the characteristics of the system that are needed for comprehension, and then present the integration in this system.

4.1 The Arias Design

Arias is a single address space operating system based on the Distributed Shared Memory (DSM) paradigm. Arias is currently under development and is being integrated in the AIX Unix system.

In Arias' memory, the unit of allocation is the *segment*. A segment is a contiguous set of pages that is dynamically allocated for applications' needs. All segments are persistent; they are shared between the Unix processes that are running in the distributed system.

The segments are managed in a single address space. At creation time, each segment is associated with a unique virtual address. Subsequently, each segment is always accessible at that virtual address, thus avoiding the need for any name translation at execution time.

Arias' segments are dynamically mapped in the address spaces of user level processes. At the first use of a segment (i.e. at the first access to a virtual address in the segment) in a Unix process, an addressing error event is delivered to the local Unix kernel. In response to this event, the kernel maps the segment in the address space of the process and then restarts the execution. The segment can be either a data segment or a code segment.

4.2 Integration of the Protection Model

Our first implementation decision is to store control informations within the Unix kernel space. This kernel space is an area in the distributed shared memory which is accessible from each machine in the system. A capability is implemented by a pair composed of access rights and a pointer to the segment. The capabilities in a domain are regrouped in a table called a domain descriptor.

The implementation of the protection model is used to control the accesses to segments by agents associated with user level processes. For this purpose:

- a protection domain is implemented by a Unix process,
- an agent is implemented by a thread in a process (a different thread per protection domain).

For efficiency reasons, access control has to take place at the first access to an object and we intend to use Unix exceptions to detect and process such an access. Initially, no segment is mapped in the virtual space of the process implementing a domain.

When an agent tries to access a data segment through a virtual address, an exception occurs. In response to this event, a capability for the segment is searched for in the domain

descriptor. If the search succeeds, the segment is mapped in the process virtual space with the rights defined in the capability. If the search fails, a protection exception is returned to the agent.

In the case of a code segment, and if the domain descriptor contains a capability with the right “execute”, then the segment is mapped in the current process in the same way as before. If a domain capability is associated with the segment, then a cross domain call is forced. We implement this call in the same way as an RPC: with each domain capability, we associate a stub for the procedure, which calls a system primitive, passing the effective parameters to the kernel. If necessary, the kernel will create a new process and a thread in this process to resume the execution of the agent. Thus, the call to the external domain is implemented by a call to a stub mapped in the current domain. The stub is “compiled” from the IDL description and is mapped at first call like any ordinary object.

5 Conclusion and Perspectives

We have presented a novel protection model based on *hidden software capabilities*. The main advantage of the model is that it separates protection definition from implementation code, thus enhancing modularity and flexibility.

This protection model defines two types of capabilities: object capabilities that allow objects to be dynamically mapped, and domain capabilities that force cross-domain invocations when protected operations are called. In order to control capability parameters exchanges between protection domains, an extended IDL is used to define a Protected Procedure Interface (PPI) associated with each domain capability. This PPI specifies the capabilities that should be passed with object name parameters, and is used to generate the code used at run-time.

From the point of view of an application developer, the main benefit of our proposal is the ability to experiment different protection schemes without any modification or recompilation of the source code. From the point of view of the system implementer, this protection model can be integrated in an operating system as a separate module; the only hook to the kernel is through the exception based detection mechanism.

This protection model can be integrated in a wide range of operating systems. For example, we have outlined its integration in the Arias single address space distributed shared memory system. Arias and our protection model are currently being implemented with the help of the facilities provided by the AIX kernel. The protection model will be evaluated in this framework.

Finally, we are also studying how an object based distributed architecture such as CORBA could benefit from this protection model.

Acknowledgments:

P. Dechamboux, J. Han, A. Knaff and E. Pérez-Cortés contributed to the design of the Arias system, including the work described in this paper. We would like to extend special thanks to S. Krakowiak for his helpful comments and careful readings of the paper.

This work was partially supported by CNET (France Télécom).

Bibliography

[Birrell 84] A. D. Birrell, B. J. Nelson. Implementing Remote Procedure Calls, ACM Transactions on Programming Languages and Systems, 2(1), February 1984.

[Cabrera 92] L.-F. Cabrera, A. W. Luniewski, J. W. Stamos. Fine-Grained Access Control in a Transactional Object-Oriented System, *Computing Systems*, 5(3), Summer 1992.

[Chase 94] J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System, *ACM Transactions on Computer Systems*, 12(4), pp. 271-307, November 1994.

[England 75] D. M. England. Capability, Concept, Mechanism and Structure in System 250, *RAIRO-Informatique (AFCET)*, Vol 9, pp. 47-62, September 1975.

[Hagimont 94] D. Hagimont. Protection in the Guide Object-Oriented Distributed System, *Proc. of the 8th European Conference on Object-Oriented Programming*, pp. 280-298 , July 1994.

[Kowalski 90] O. C. Kowalski, H. Härtig. Protection in the BirliX Operating System, *Proc. of the 10th International Conference on Distributed Computing Systems*, pp. 160-166, May 1990.

[Levy 84] H. M. Levy. *Capability-Based Computer Systems*, Digital Press, 1984.

[OMG 91] OMG. The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1, Revision 1.1, December 1991.

[Organick 72] E.I. Organick. *The Multics System: an Examination of its Structure*, MIT Press, 1972.

[rpcgen 88] rpcgen - An RPC Protocol Compiler, Sun Microsystem, Inc., 1988.

[Shapiro 86] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle, *Proc. of the 6th International Conference on Distributed Computing Systems*, pp. 198-204, 1986.

[Tanenbaum 86] A. S. Tanenbaum, S. J. Mullender, R. Van Renesse. Using Sparse Capabilities in a Distributed Operating System, *Proc. of the Sixth IEEE International Conference on Distributed Computing Systems*, pp. 558-563, 1986.

[Wilkes 79] M.V. Wilkes, R. M. Needham. *The Cambridge CAP Computer and its Operating System*, North Holland, 1979.

[Wulf 74] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack. Hydra: The Kernel of a Multiprocessor Operating System, *Communications of the ACM*, 17(6), June 1974.