# Protection Reconfiguration for Reusable Software

Christian Damsgaard Jensen* Daniel Hagimont†
INRIA Project SIRAC
655, avenue de l'Europe
38330 Montbonnot – France
Christian.Jensen@imag.fr

## Abstract

*Secure distributed applications often include code to authenticate users, verify access rights, and establish secure communication channels between software components (e.g., clients and servers). This code is often specific to the application and to the context in which the application is used. Embedding protection definitions in the application code makes it difficult to reuse because different applications often have very different protection constraints. In this paper we argue that protection definitions should be specified separately from application code in order to facilitate protection reconfiguration and software reuse. This separation may be achieved by specifying protection definitions in an Protection Interface Definition Language, and is implemented by proxies installed between software components that are executing in different protection contexts (e.g., between client and server). Mechanisms for creation and dynamic installation of protection proxies have been implemented in a distributed shared memory system, and the viability of our solution is demonstrated by a simple name server, which is currently being reused in three different protection contexts.*

## 1. Introduction

Recent advances in software engineering, such as the notions of middleware [2] and software architectures [9, 10, 8, 1], allow programmers to compose applications from a mixture of existing software modules, third party libraries, legacy programs, and a minimal amount of code developed for that particular application. However, most of these efforts ignore application protection leaving it to the underlying operating system, which is inadequate when applications are composed from components of different provenance.

The security problem of reusing code in distributed applications is probably best illustrated by the use of external programs in most WWW-browsers. Most browsers can be configured to use external/third party programs to display images or PostScript files, play music or video clips, convert MIME encoded mail, or simply allow users to specify external pagers and editors. These programs are potential Trojan horses, because they rarely implement a protection policy that allows them to verify data before operating on them. Most of these external programs are developed to be used in safe environments, e.g., to allow users to preview their private PostScript documents (PostScript is a full programming language, that for instance allows programs to access files in the file system), but are being reused by the WWW-browser in the potentially hostile environment of the Internet, where data may be controlled by an adversary. In the example of the PostScript previewer all protection constraints are met. However, a previewer that implements the full PostScript definitions, must include primitives that allow access to the file system, which allow potential intruders to install a Trojan horse from a PostScript document.

Bugs and undocumented features are well known ways of breaching a systems security, but the example above shows that programs that comply with specifications can be just as dangerous. All software components should therefore be considered untrusted, and executed with the least possible privileges; this is also known as confinement or the need–to–know principle.

In this paper we propose a model where each reused software component is confined to a restricted execution context (often called a sandbox), where it can do little harm if protection constraints are violated. In order to facilitate software reuse, protection constraints are managed in the interfaces between software components. This allows protection specifications to be altered by redefining the interfaces (and recompiling the stubs), without modifying the components themselves.

The rest of this paper is organized as follows: section 2 defines the motivation for this work including the threat

* Université Joseph Fourier
† INRIA Rhône–Alpes

model. We propose a protection model for reusable software in section 3. This model has been implemented in a distributed shared memory system, and an experiment with software reuse in this system is described in section 4. We review related work in section 5. Section 6 presents our conclusions and describes some directions for our future work.

## 2. Motivation

In the following we define our model for software reuse and the threat model we are considering, in order to establish the protection requirements for software reuse. We mainly focus on application protection and access control, and do not consider problems relating to safety of applications (correct execution), nor problems relating to overuse of resources (denial of service attacks).

### 2.1. Software Reuse Model

An application programmer should be allowed to reuse software components without knowing all the protection constraints of each component, and to tailor the known protection constraints to the needs of the application. This leads to the following model:

> *Components* are computational units written in any programming language. The protection constraints of components are not necessarily known, so they are generally considered untrusted. Components are the bricks of an application.
>
> *Connectors* define the interactions among components — and allow explicit protection constraints to be met, when there exist a valid mapping between the respective constraints of the interacting components. Connectors are defined in a Interface Description Language (IDL). Connectors are the mortar of an application.
>
> *Configurations* define the application structure through component interconnections. The configuration also defines the protection context of each component, which is described further in 3.3. The configuration constitutes the application built from components and connectors.

In order to define an appropriate protection model, we first need to examine the protection problems introduced by the above model. This is done in the following sections.

### 2.2. Threat Model

From the application programmers point of view, software components should be black boxes that promise to provide certain semantics for well defined interfaces. However, there is no guarantee that the black box actually keeps that promise, or that it does not attempt to compromise the security of the application, the machine running it, or the network segment hosting that machine.

A component that executes with all the privileges of the user launching the application may accidentally or maliciously corrupt, or delete data (e.g., files) belonging to that user or leak sensitive data available to that user. The component should therefore be executed with the strict minimum of privileges following the need–to–know principle.

### 2.3. Protection Requirements for Software Reuse

Most software components are written with both explicit protection requirements and implicit security constraints. The requirements are normally well documented, and include data encryption and establishment of secure communication channels, authentication of remote entities, and explicit passing of credentials to access control mechanisms. The implicit security constraints are more elusive, often involving bounds on parameters and violation of consistency of data structures, e.g. in a procedural environment, caller and callee must agree on the size and format of parameters passed between them. Secure reuse of software components requires that both protection requirements and security constraints be met.

Different applications have different protection requirements, e.g., communication between a client and a server behind the corporate firewall can be passed in clear text, while connections from a mobile user calling through a public network needs to be encrypted if it carries sensitive information. Reconfiguration of protection requirements are therefore an important aspect of the software reuse process.

The implicit security constraints of a software component are by nature difficult to know in advance. We therefore wish to confine these components to a restricted execution environment (a sandbox), where access to system resources is limited to a strict minimum. It is important to note that components executing in their sandbox cannot execute with the full privileges of the user who launched the application.

## 3. Protection Model

The protection model described in the following was originally developed for a distributed shared memory system called Arias [6, 4]. Our preliminary experiences with applications written for Arias and discussions with colleagues who develop the Olan distributed application configuration language [1], has convinced us of its general applicability for software reuse.

### 3.1. Access Control

The protection model follows the general capability model [7], and is based on the notions of capabilities, protection domains, and cross domain calls.

> *Capabilities* are tokens that identify a resource and specify a set of allowed operations. Anyone who whishes to access a protected resource has to present a capability for that resource. Capabilities can be copied (possibly with reduced access rights) and passed along to somebody else, providing them with access to the resource.
>
> *Protection domains* define the protection contexts of software components (i.e., the sandboxes). The access rights of a protection domain are determined by a list of capabilities maintained for that domain.
>
> *Cross domain calls* allow applications to call well defined *entry points* in components executing in another protection domain. The interfaces of these entry points are defined in the Protection IDL, that extends the IDL with protection statements. The PIDL is described in greater detail in 3.3. The PIDL definitions are compiled into client and server stubs, that allow controlled transitions among software components.

Components are confined in protection domains, and allowed to directly access resources for which a capability exists in the protection domain. A capability may be associated with either data or code; it grants read and/or write access to data, and execute or cross domain call permissions for code. Capabilities that allow components to perform a cross domain call are simply called domain capabilities.

Components interact through protection interfaces, which are specified separately for caller and callee in order to support mutual suspicion between software components. The pair of stubs compiled from these interfaces constitute the connector between the two components.

A configuration is determined in terms of capabilities and protection domains. Components are identified and a minimal protection domain is defined for each component. This domain must contain access capabilities for data that are directly accessed by the component and domain capabilities for the components with which it interacts. The mechanism that allows safe transitions between protection domains is described in 3.3.

### 3.2. Protection Domains

The protection domain define the access rights of the component. It contains a list of initial capabilities held by that domain, and a list of capabilities that have been added to the domain in order to temporarily augment its privileges.

The ability to dynamically add capabilities to a protection domain means that the initial capability list of a protection domain can be kept minimal, and new capabilities passed along with parameters as needed. In the example of the PostScript previewer, this would mean that all access to the file system can be initially denied. When the previewer is launched, a read capability is passed along with the file name parameter allowing the previewer to access this file only. Capabilities are normally added to a protection domain as a result of a cross domain call, and revoked when the call returns or in case of failure.

### 3.3. Protection Interfaces and Cross Domain Calls

A protection interface defines the protection constraints of interactions between software components (cross domain calls). It can control the way components interact in two ways: either by specifying the means of interactions, e.g., data encryption using a secure RPC mechanism, or by defining the transfer of access rights between the two domains, e.g., when a filename is passed as parameter, rights to access that file should be transferred as well.

The protection interface definition identifies the entry point, i.e, the name of the procedure to call in the remote protection domain. Furthermore, it describes the parameters given as input or expected as output and the capabilities that must be passed along with these parameters.

For all parameters to an entry point, the protection interface must specify the type and whether the parameter is passed as input (`IN`), output (`OUT`), or both(`INOUT`). In addition to the direction of parameters, the protection interface must also define any capabilities that are to be passed along with a parameter, these capabilities can be of any type, i.e., read (`CAPA_READ`), write (`CAPA_WRITE`), or cross domain call (`CAPA_DOM`). Examples of protection interfaces are shown in 4.2.1, 4.2.2, and 4.2.3.

The protection interfaces between all components are specified in the configuration of the application and are transparent to the programmers of the components. Each programmer simply assumes that the necessary capabilities are passed along with parameters when entry points in the component are called, and that the necessary capabilities will be passed along with the parameters when entry points in other components are called. An identification of the protection interface between two interacting components is embedded in the capabilities that allow the components to interact, i.e. at execution time, the capabilities in the current protection domain decide which protection interface should be used.

The task of configuring an application consists of specifying the protection interfaces, creating the different protection domains, and adding the appropriate capabilities to the protection domains.

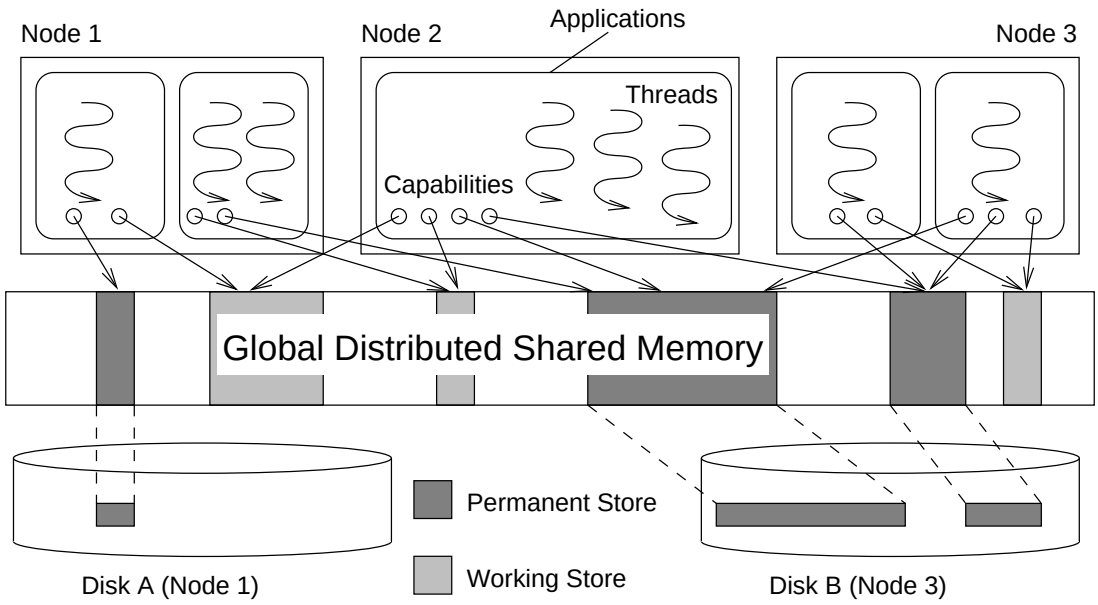


**Figure 1. Overview of the Arias Distributed Shared Memory**

## 4. An Example of Protected Reuse

In the following we describe our experiments with different protection policies for a single software component. The experiments were performed in a distributed shared memory system called Arias, that extends the shared memory notion to a network of loosely coupled workstations.

### 4.1. The Arias Distributed Shared Memory

In the following we describe Arias in order to define the context of our experiments.

#### 4.1.1 Overview

The Arias DSM defines a persistent global address space where objects are uniquely identified by their virtual addresses. The architecture of the Arias system is illustrated by figure 1.

The figure shows The Global Distributed Shared Memory in a network of three nodes (workstations). Node1 and Node3 both have disks, where permanent copies of regions of shared memory are stored. Each application running on one of these nodes holds a number of capabilities that allow it to access regions of shared memory.

The unit of shared memory allocation in Arias is the *segment*, which is a contiguous set of virtual memory pages identified by the base address. The size of a segment is fixed when the segment is created, and cannot subsequently be changed. Segments are persistent, which means that objects created in the shared memory exist until the end of time or explicitly destroyed. Persistent segments can be made permanent by storing them to disk, this prevents persistent objects from saturating physical memory by saving them on stable storage, and freeing their physical memory. Permanent segments are brought back into physical memory by demand paging.

Segments may be used to store either data or code. Data segments provide applications with a single level store, and work much like memory mapped files (they are always mapped at the same address and do not require an explicit `open` operation). Software components are developed using the system compilers (C, Ada, Fortran, COBOL) and the binary object is then loaded into a code segment created for that component.

#### 4.1.2 Access Control

Access to segments of shared memory is controlled by capabilities, that are verified the first time a component tries to address a segment. All management and verification of capabilities is transparently performed by the system.

For data segments the system verifies that a capability for the segment exists in the protection domain of the component, if it does the segment is mapped into the address space of the component process, otherwise a protection fault occurs.

For code segments the system first checks whether an execution capability exists for that segment (component), if it does the segment is mapped into the address space of the component (this is the method for constructing composite components) and external references are resolved, other-

wise the system searches the protection domains for a domain capability. If the system finds a domain capability, it maps a client stub in the place of the real component to perform a cross domain call. If no capability is found, a protection fault occurs.

#### 4.1.3 Execution Model

In general, Arias applications are launched from Unix and execute as one or more Unix processes. All processes that use the shared memory reserve the same large portion of their virtual address space for Arias; the size of this reservation is a parameter that is fixed when the Arias system is installed. Entry points in components are accessed through simple procedure calls, that the system transforms into cross domain calls when the protection domain holds a domain capability instead of an execution capability.

### 4.2. Reusing a Simple Name Server

We have implemented a simple name server library that maps symbolic names to virtual addresses (unique object references). This name server exports three entry points: *create*, *add*, and *lookup*, that respectively allow creation of a new name server (allocation of shared memory and initialization), addition of new entries, and looking up names in the name server. Prototypes in C for the name server functions are shown in figure 2.

We used this name server to experiment with implementation of different protection policies for the same software component.

#### 4.2.1 Application Specific Name servers

Most distributed applications contain a name server in some form or other, to map symbolic names onto physical system entities (network servers, files, ...). The name server is an integrated part of the application that is allowed to manipulate all entries freely. If all implicit protection constraints of the name server are known (it is a simple component developed in house), an execute capability can be included in the protection domain of the caller, which effectively links the library with the application. Otherwise, a protection domain must be created and the name server called in that domain (note that the protection interface allows all operations and directly maps parameters).

#### 4.2.2 Protected Name Server

A system wide name service generally restricts access to its internal state. The protected interface shown in figure 3(b) allows clients to add entries and search entries, but not to create new name servers. Furthermore, by restricting access to name server data to well defined functions (specified in the interface), the consistency of the internal state is ensured. In the example, clients are allowed to add entries, but an error is returned if the name already exists in the name server.

The semantics of this protection scheme is similar to that of most object–oriented systems, i.e., internal data can only be manipulated through exported operations.

#### 4.2.3 Combined Name and Protection Server

The name server described above allows components and applications to exchange references to data in the shared memory, but not necessarily to actually share data. In order to access data, the protection domain must hold a capability for the corresponding segment of shared memory. In order to facilitate sharing of data, the protection interfaces have been extended to pass capabilities between protection domains. We simply add the key word CAPA_READ following the direction clause, which specifies that a read capability should be passed along with the parameter. When a component registers a name and address in the name server, a read capability is automatically passed to the name server domain. This capability will be copied and passed along to any component querying the name server for that name. The interface used in the combined name and protection server is shown in figure 3(c).

#### 4.2.4 Conclusions of Experiment

The separation of protection definitions and algorithmics allows us to implement highly different protection policies for the same software component. Furthermore, it allows us to manage all explicit protection constraints in the protection interfaces, without modifyingthe code of the application.

## 5. Related Work

We have found no other work that explicitly targets the problem of protection in software reuse. However, there are two current projects that work on ideas similar to ours (interface–based protection programming or protection encapsulation of existing software).

The idea of specifying protection constraints in the interfaces of software components is being investigated by the Solidor project at IRISA, Rennes [3]. Their work focuses on specifying security policies for the interactions among components, i.e., encrypted communication and authentication of remote entities. In this respect, their work complements the work presented in this paper.

The Janus system developed at Berkeley [5] allows untrusted programs to be confined, e.g., the WWW-browsers external helper programs. The system works by filtering system calls issued from a program launched under Janus,

```
void create(addr_t *ns, int elem_size, int no_elem);
void add(char *name, addr_t *addr);
void lookup(char *name, addr_t *addr);
```

**Figure 2. Name server prototypes**

```
void create(OUT addr_t ns, IN int elem_size, IN int no_elem);
void add(IN char *name, OUT addr_t addr);
void lookup(IN char *name, OUT addr_t addr);
```

(a) Application specific name server

```
void add(IN addr_t ns, IN char *name, IN addr_t addr);
void lookup(IN addr_t ns, IN char *name, OUT addr_t addr);
```

(b) Protected name server

```
void add(IN addr_t ns, IN char *name, IN CAPA_READ addr_t addr);
void lookup(IN addr_t ns, IN char *name, OUT CAPA_READ addr_t addr);
```

(c) Combined name and protection server

**Figure 3. Protection Interfaces for the name server**

and only allows those calls according to a predefined protection policy, e.g., the system call `open()` is only allowed for explicitly allowed files. The protection policy is defined in configuration files, and lacks the dynamism provided by the use and transfer of capabilities in Arias. Furthermore, Janus confines entire programs and does not allow different program components to be executed with different access rights.

## 6. Conclusions and Perspectives

In this paper we have presented a protection model for reusable software components. This model separates protection definitions from application code by managing protection specifications in the interfaces between software components. This allows protection specifications to be changed without modifying the components themselves.

Our implementation of this model in Arias is based on capabilities and the ability to transfer capabilities along with parameters in procedure calls. This allows us to define protection domains with few initial capabilities and dynamically add capabilities to that domain as needed in order to enforce the need–to–know principle.

Our experiment shows that protection definitions can be successfully separated from the application code, which allows multiple protection policies to be specified for the same software component.

We therefore believe that this is a convenient protection model for reusable software, and note that the model is general, i.e. there are no inherent dependencies on the original DSM system.

A distributed shared memory system is a restricted platform for reusable software. We are therefore working on a prototype, that implements the Arias protection model in a CORBA compliant Object Request Broker (Orbix [11]). This implementation will allow us to experiment with a number of extensions to the PIDL, notably concerning authentication and communication security.

### Acknowledgments

We would like to thank our colleagues working with the Olan configuration language for the many fruitful discussions that inspired us to write this paper.

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

## References

[1] L. Bellissard, S. BenAtallah, F. Boyer, and M. Riveill. Distributed application configuration. In *16th International Conference on Distributed Computing Systems*, pages 579–585, Hong Kong, May 1996.

[2] P. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.

[3] C. Bidan and V. Issarny. A configuration-based environment for dealing with multiple security policies in open distributed systems. In *2nd European Research Seminar*

*on Advances in Distributed Systems*, pages 240–245, Zinal, Switzerland, March 1997.

[4] P. Dechamboux, D. Hagimont, J. Mossière, and X. Rousset de Pina. The Arias distributed shared memory: An overview. In *SOFSEM '96*, number 1175 in Lecture Notes in Computer Science, pages 56–73. Springer Verlag, 1996.

[5] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *6th USENIX Security Symposium*, pages 1–13. USENIX, July 1996.

[6] D. Hagimont, J. Mossière, X. Rousset de Pina, and F. Saunier. Hidden software capabilities. In *16th International Conference on Distributed Computing Systems*, pages 282–289, Hong Kong, May 1996.

[7] H. M. Levy. *Capability–based Computer Systems*. Digital Press, Bedford Massachusetts, 1984.

[8] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *5th European Software Engineering Conference*, pages 137–153, Sitges, Spain, September 1995.

[9] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[10] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[11] I. Technologies. *Orbix distributed object technology – programmers guide, Version 1.2*. Iona Technologies, 1994.