CHAPTER 5

A Configurable Security Architecture Prototype

Alexandre Hardy
ah@adam.rau.ac.za

Martin S Olivier
molivier@rkw.rau.ac.za
Department of Computer Science
Rand Afrikaans University
PO Box 524, Auckland Park, Johannesburg, South Africa

Abstract Traditional security systems are integrated closely with the applications that they protect or they are a separate component that provides system protection. As a separate component, the security system may be configurable and support various security models. The component does not directly support the application. Instead, operating system objects (such as files) are protected. Security systems that are integrated with the applications that they protect avoid this shortcoming, but are usually not configurable. They also cannot provide the same level of protection that a system provided security component can enforce, as the application does not have access to the hardware that supports these features. The Configurable Security Architecture (ConSA [1]) defines an architecture that provides the flexibility of a system security component while still supporting application security. Such an architecture provides obvious benefits. Security policies can be constructed from off-the-shelf components, supporting a diverse array of security needs. Before this or a similar architecture can be accepted by the industry, the concept must be proven to work theoretically and practically. Olivier [1] has developed the theoretical model and illustrates its usefulness. This paper describes an implementation of ConSA and in so doing, proves that ConSA can be implemented in practice.

Keywords: Access Control, Security, Security Model, Prototype

1. INTRODUCTION

An architecture that supports arbitrary security policies through the use of off the shelf components will greatly simplify the implementation of security systems. Such an architecture will allow developers of security models to concentrate on the access control algorithms of such a model, without concern for the method in which access control will be enforced. The ConSA (Configurable Security Architecture) architecture [1] is one such system. Olivier [1] describes the model and provides a formal description of the various components. The model will not be described again in this paper, due to lack of space. Section 2 will however briefly illustrate how ConSA functions. This paper will instead describe a proof of concept prototype that illustrates that ConSA can be implemented on existing systems. The prototype is implemented in the Linux operating system, and can be moved to another of the many UNIX [9, 11] like operating systems available. Furthermore, the prototype illustrates the use of ConSA for application level security and system level security.

In section 2 a brief background to security is presented, section 3 discusses the prototype and the three significant changes to the ConSA model: the ConSA kernel, Protect Table and Message Dispatcher. Section 4 presents the conclusions drawn from the prototype.

2. BACKGROUND

A security policy determines who may access (and who may not) objects or entities in a system. A security system on a computer must enforce such a policy, ensuring integrity and availability of data. The process of enforcing the security policy is known as *Access Control.* Several security models have been proposed that may be used to implement certain security policies. These models have generally been classified as *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC - also known as multilevel access control).

Discretionary Access Control associates an owner with each object in the system. The owner may then grant (or revoke) access to (from) other subjects in the system so that they may access (may not access) that object. The object may have an *Access Control List* associated with it that contains a list of all subjects that may access that object.

Mandatory Access Control associates a clearance level with each subject and a classification with each object. A subject may obtain read access to an object if the clearance of the subject dominates the classification of the object (known as the *simple security property*). Write access may be obtained if the clearance of the subject is lower than the classification of the object (known as the *star property*). The simple

security property prevents unauthorized subjects from viewing sensitive information, while the star property prevents subjects of high classification from making sensitive information available to subjects of lower classification.

Flow control ensures that if data is written to an object, then all the subjects who could not access the data in its previous location will still be unable to do so. Further information on security models and other aspects of security can be found in [3, 4, 5, 6].

The concept of *labels* can be applied to most security models. A label encodes security information and may be associated with subjects or objects and perhaps others. Labels are well suited to implementing more advanced and dynamic security models [7, 8].

Traditional security systems have only implemented one security model, ConSA provides a security architecture that may be used to support and enforce a variety of security models. We refer to [1, 2] for details on the ConSA model, and only provide a short description here.

The ConSA system has two types of labels, *Entity Labels* protect objects and determine who may access those objects. *Subject Labels* are tokens presented by subjects that may grant them access to certain objects. The *Subject Management Module* determines which Subject Labels a subject will use. The *Authorization Control Module* determines the initial protection attributes of an object that has just been created. The Protect Table associates Entity Labels with the objects that they protect and controls access to these objects. The *Message Dispatcher* is responsible for relaying messages between objects and enforcing the security model on those messages. And finally, the *Information Flow Manager* enforces flow control on messages.

Typical system interaction may occur as follows: A subject logs onto the system and is presented with a subject label by the Subject Management Module. When the subject sends a message to an object, the Message Dispatcher determines which label protects the object by consulting the Protect Table, and then determines if the message will be relayed by consulting the Entity Label and Information Flow Manager.

3. THE PROTOTYPE

The most important modifications to the ConSA model for the prototype are the introduction of a kernel component that controls the ConSA system, modifications to the role of the Protect Table, and a specification for the Message Dispatcher. The kernel component allows for the specification of how the various ConSA components interact. This interaction is controlled by the ConSA kernel and can be specified by

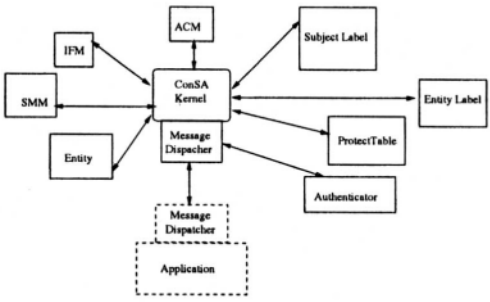

Figure 1. Prototype ConSA architecture

```
Class Entity
Public
   Create(resource)
   Destroy(resource)
   Open(resource,mode)
   Read(data,len)
   Write(data,len)
   Control(cmd,arg)
   Close()
```

Figure 2. Entity interface

sequential algorithms. The Protect Table and ConSA kernel describe how the task of the original Protect Table may be achieved. Lastly the Message Dispatcher is specified so that interaction between applications and the security system may be achieved. Space restrictions preclude a discussion of the other components of the ConSA architecture.

3.1. THE ConSA KERNEL

The ConSA model allows various modules to work together to determine and enforce a security policy. It is clear that a Message Dispatcher component is needed to transfer messages between the ConSA system and user programs. It will be useful to define a new component that integrates the existing components, and defines how these components interact. This component shall be referred to as the ConSA kernel. Now the only concern of the Message Dispatcher is to facilitate the passing of messages between components. The ConSA kernel will handle requests and delegate to the necessary components. In this way, only the Message Dispatcher need be rewritten to support new message transferring methods. These may include network transports, API calls or system calls.

The system can be implemented at different levels to increase the inherent security or efficiency of the system. For example a Message Dispatcher in the form of a linkable library (including the rest of the ConSA system) would increase efficiency. A library would be supplied that clients may use to access the ConSA system. This also allows the system to be easily subverted. Another option would be to integrate the ConSA system into the Linux kernel so that existing system routines may be protected by ConSA (and perhaps more). This approach is slightly less efficient, but inherently much more secure, since the kernel

memory is inaccessible to user space programs. Now some component must decide whether messages are to be permitted or denied. The Entity Label module and Information Flow Manager both have an effect on the outcome of this decision. Modularity of the ConSA system would be improved if these two modules did not have to communicate in order to reach a decision. This can be achieved if we allow the ConSA kernel to make the decision, with 'advice' from these two modules. This process will be illustrated a little later.

Furthermore it is difficult to allow the Protect Table to intercept messages to enforce security concerns. The result is that the ConSA modules in the prototype are trusted. The modules themselves are responsible for correctly implementing the security model when inter-module communication occurs. The ConSA kernel will however facilitate this to some extent. Figure 1 illustrates the architecture selected for the prototype.

The Message Dispatcher is responsible for communication between the user and the ConSA system, and the Protect Table simply remembers which labels protect which entities. The ConSA kernel accepts messages from the Message Dispatcher and coordinates with the other modules to achieve the desired result.

3.1.1 Entities. Arbitrary messages are usually difficult to intercept or not very efficient. These problems may be simplified by defining a communication interface for objects, that can be easily monitored and still support arbitrary messages. To support the Message Dispatcher and facilitate message passing to Entities, an Entity interface is defined, as illustrated in figure 2. The Entity Label and Subject Label do not have to conform to this specification. This Entity interface is required for resources in the system to be accessed uniformly by the ConSA kernel. Arbitrary messages can be implemented at a higher level by using this interface as a transport.

The methods listed in figure 2 are self explanatory. Resources are accessed using a resource locator format of *class: resource*. For example *file: example.txt* refers to an entity of class *file*, and the resource to be accessed is *example.txt*. Now that entities have been defined the services that the ConSA kernel provide can be examined.

3.1.2 ConSA Kernel Services. There are a large number of services defined to support module interaction and internal administrative tasks. A discussion of all these services is beyond the scope of this paper, and cannot be presented due to space constraints. The services that implement the ConSA interface are of much greater interest, and so only these services will be discussed. Once again, only selected services

```
function KOpen(token,resource,mode)
      SubjectLabel subject;
      EntityLabel label;
      InformationFlowManager IFM;
      if (InValidToken(token)) then return FAILURE;
      subject=SubjectLabelOf(token);
      IFM=IFMAssociatedWith(subject);
      label=GetLabel(resource);
      if (label is not valid) then
            if (policy=DENY) then return FAILURE;
      if ((label is valid) and
                  (not label->CheckAccess(subject,mode))) then
            return FAILURE;
      IFMresult=IFM->ActivateFlow(FLOW_OPEN,GetLabel(resource),
                  subject,FALSE);
      if (IFMresult is FAILURE) then return FAILURE;
      return OpenEntity(resource);
end function
```

Figure 3. KOpen algorithm

will be presented as a result of space restrictions. Please refer to [2] for details on all these algorithms. The `KOpen` and `KGrantAccess` algorithms will be presented as they are fairly representative of the structure of these algorithms.

`KOpen (subject, resource, mode)`
The `KOpen` service opens a resource for access specified by `mode`. This service must ensure that the specified subject has been granted the required access to the resource. The Protect Table is enlisted to determine which labels protect the requested resource. The algorithm is listed in figure 3.

The first action taken is to validate the subject token provided, by attempting to locate the associated subject label and Information Flow Manager. Next the label protecting the entity, and specifically the open method (if any) is located via the `GetLabel` call. If no suitable label is found, then the default policy of the ConSA system determines if access is granted or not. If the label permits access, then the Information Flow Manager is consulted. In this service the Information Flow Man-

```
function KGrantAccess(token,resource,user_for_ACL,mode)
      SubjectLabel subject;
      SubjectLabel usertoadd;
      EntityLabel label;
      EntityLabel protlabel;
      InformationFlowManager IFM;
      if (InValidToken(token)) then return FAILURE;
      subject=SubjectLabelOf(token);
      IFM=IFMAssociatedWith(subject);
          label=GetLabel(resource);
      //If there is no label:  default policy would
      //determine if access is granted or not
      if (label is not valid) then
            if (policy=DENY) then return FAILURE;
            else return SUCCESS;
          protlabel=GetLabel(label);
      //IFM is consulted first, for proper
      // logical and construction
      if (not IFM->ActivateFlow(FLOW_GRANT,protlabel,
                  subject,TRUE)) then return FAILURE;
      if (protlabel is not valid) then
            //Policy determines if grant will take place
            if (policy=ALLOW) then begin
                  usertoadd=SMM->Translate(user_for_ACL);
                  return label->GrantAccess(usertoadd,mode);
            end else
                  return FAILURE;
      if (not protlabel->CheckAccess(subject,MODE_GRANT)) then
            return FAILURE;
      else begin
            usertoadd=SMM->Translate(user_for_ACL);
            return label->GrantAccess(usertoadd,mode);
      end
end function
```

Figure 4. KGrantAccess Algorithm

ager cannot deny the request, and still indicate that the operation was successful.

`KGrantAccess (subject, resource ,user_for_ACL, mode)` **(figure 4)**
It must be possible for the System Security Officer, and perhaps for other subjects (as in Discretionary Access Control) to change the access control list associated with a resource. The entity label provides these services, but the application cannot communicate directly with the label. The ConSA kernel provides access to the entity label, and also determines whether the subject may in fact access the entity label (which is also a resource). For access to the ACL, the Information Flow Manager is consulted first. It does not matter in which order the Information Flow Manager and entity label are consulted. The operation is always the logical **and** of the results. It is important to see that the label protecting the resource is the entity which is being checked. Once the label has been found, the label protecting this label is in turn found. This label is then the label that determines if access to modify, or read the label is granted or not. This label is the variable `protlabel` in the algorithm. The Subject Manager provides the subject label for the subject that is to be queried, added or removed from the ACL. The user only knows some token that identifies the subject. The entity label may be modified to accept extra parameters specifying which type of access is to be granted (to pass to the `protlabel` variable in this case). Another simpler alternative may also be followed, specific modes of access combining say granting and reading, may be combined to form a new mode: grant read access. If numerical values are used for modes, a bitwise or operation may perhaps be used to obtain the new mode. A further question to ask is, how may we implement Discretionary Access Control with this algorithm? To allow subjects to grant access, the `protlabel` needs to be modified. The algorithm does not modify this label at any stage. Two options are however possible:

- Construct a new resource for accessing and modifying labels. This new resource will allow subjects to query and modify labels at all levels.
- The label protecting the object can determine which label protects it. If the label is to grant access for grant mode then the label simply grants access to the label protecting it. Some method must be developed to differentiate between a label requesting the grant access and another portion of the program from requesting grant access, otherwise the label will recursively grant access to the parent.

3.2. PROTECT TABLE

Entity Labels protect objects in the system. The Entity Labels cannot practically be stored with the objects they protect due to the diversity of the security models that may be implemented. Some other method must be devised to associate objects with their Entity Labels. One solution is a Protect Table that associates labels with objects. The entire table is stored on secondary storage in some representation that may be implemented independently of the file system. This also allows queries to quickly locate labels associated with an object, without first locating the object. The next section lists the services required from a Protect Table.

3.2.1 Protect Table Services. One method has been identified in [1] for the Protect Table, namely `Protect. Protect` associates an Entity Label with an object. The association means that the specified Entity Label controls access to the object it is associated with. It is assumed that at most one Entity Label will protect an object. If two labels L_1, L_2 could protect an object then we could construct a single new label L_3 such that $subj(L_3) := subj(L_1) \bigcap subj(L_2)$[1]. If the label module cannot support this, then a new label module may be constructed that can implement the new label. This specification is relatively simple, but in a practical implementation there are several difficulties. It is difficult to capture attempts to access an object. A universal communication scheme may be implemented to help trap messages, but this cannot be enforced. One object may be instantiated several times in the system, and may have a different identifier at each instantiation. The identifier is provided by the operating system. The information the Protect Table stores must not be lost if the system is halted.

The prototype attempts to address these problems by enforcing protection with the ConSA kernel and Message Dispatcher. A new object identification scheme was implemented to provide objects with identifiers that remained the same even after system reboot. The Protect Table has also been extended to explicitly support protection of objects, classes and methods. The services implemented are listed in table 1.

3.3. MESSAGE DISPATCHER

The Message Dispatcher is responsible for forwarding messages to the correct objects, and enforcing the selected security model on these messages. It is very difficult to intercept messages in an already existing system. A better approach is to implement a new message transmission technique that is difficult to circumvent, and to make services available

Table 1. Protect Table extended methods

Service	Parameters
`ProtectClass`	`(Class,Label)`
`ProtectObject`	`(Object,Label)`
`ProtectClassMethod`	`(Class,Method,Label)`
`ProtectObjectMethod`	`(Object,Method,Label)`
`GetLabel`	`(Object,Class,Method)`

through this system. The efficiency of such a system is also important. The Message Dispatcher has no specific methods that can be identified, rather it is an inherent part of the ConSA system, and facilitates all communication between ConSA and the application. As such, the implementation is very dependent on the system selected for implementation. For the prototype, three message transports were implemented offering various degrees of security and efficiency:

- Dynamic Link Library - As a library, the ConSA system is linked into the application at compile time. The application uses the ConSA system to enforce a security policy. The application may choose not to use the ConSA security system, and this cannot be prevented. Pluggable Authentication Module in Linux [10] is implemented in a similar fashion.

- Device Driver - The device driver developed for Linux supports communication with the Message Dispatcher. The security of the communication mechanism is a lot higher. The device driver is a compiled object module that can be inserted into the Linux kernel at run time. A character device can then be created in the file system, through which communication occurs. Two applications are now required, a server application that responds to requests on one device, and a client application that communicates with the server program. The client application may still access other system resources, but the Linux security system may be used to limit access so that the character devices are the only valid means for communication.

- Kernel Integration - The ultimate level of security is integrating the ConSA system with the operating system. The messages may then be implemented in the same way as existing operating system calls are implemented, and existing system calls may be intercepted to implement the security model on all aspects of the operating

system. Now the ConSA system can also protect files that define the behavior of the system. In the prototype, the Linux syscall table was replaced to intercept system commands.

The prototype system implements each of these transports, which demonstrates the flexibility of the architecture. To provide a uniform communication system, whatever the transport, the following services have been identified for implementation:

- Message Dispatcher maintenance routines
  `MessageDispatcherInit()`
  `MessageDispatcherClose()`
  `DispatchMessage()`
  The `DispatchMessage` method is provided so that Message Dispatchers that are not automatically aware of messages, and cannot automatically dispatch the messages, may poll for messages and dispatch them if any are found.

- ConSA Kernel access routines

The ConSA kernel services are used extensively in the Message Dispatcher. By identifying these services, communication can be strictly controlled, and may be more efficient than supporting arbitrary method invocations directly. The communication with an Entity could also have been implemented by writing a device driver for each entity, but this would complicate development of entities. Instead the goal was to keep development of all modules in the ConSA system relatively transparent and independent of the level of implementation selected. This has been achieved in the prototype to a large extent. There are very few items that need replacing if the transport is changed; the various Message Dispatchers can be changed with ease to support the desired transport.

4. CONCLUSION

A prototype implementation of the ConSA system proves that the ConSA is a viable security architecture. Solutions to many of the problems with an implementation have been found. Such a system will be able to implement many security models, including newer techniques such as those presented in [8].

Notes

1. *subj*(*L*) Denotes all the subjects that the label *L* grants access to.

References

[1] M. S. Olivier, *Towards a Configurable Security Architecture*, Data & Knowledge Engineering, To appear

[2] A. Hardy, *An Implementation and Analysis of the Configurable Security Architecture, Masters dissertation*, Rand Afrikaans University, 1999

[3] S. H. von Solms and J. H, P. Eloff, *Information Security*, Rand Afrikaans University, 1998

[4] D. E. Bell and L. J. LaPadula, "Secure computer system: unified exposition and Multics interpretation", *Rep. ESD-TR-75-306*, March 1976, MITRE Corporation

[5] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations", *Secure Computer Systems: Mathematical Foundations (Mitre technical Report 2547, Volume I)*, March 1973, MITRE Corporation

[6] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: A Mathematical Model", *Secure Computer Systems: Mathematical Foundations (Mitre technical Report 2547, Volume II),* May 1973, MITRE Corporation

[7] L. Gong and X.Qian, "Enriching the Expressive power of Security Labels", *IEEE Transactions on Knowledge and Data Engineering, 7(5)* , October 1995

[8] S. N. Foley, L. Gong and X.Qian, "A Security Model of Dynamic Labeling Providing a Tiered Approach to Verification", Technical Report SRI-CSL-95-15, SRI International, 1995

[9] *The Single UNIXR Specification, Version 2*, The Open Group, 1997, www.*opengroup.org*

[10] Andrew G. Morgan, *The Linux-PAM System Administrators' Guide,* (Distributed with the PAM software package), 1998

[11] Chris Hare, Emmett Dunlaney, George Eckel, Steven Lee, Lee Ray, *Inside Unix*, New Riders Publishing, 1994