Cryptographic Access Control In Remote Procedure Call

Henrik Christensen (s991571)

Jonas Høgh (s991249)

11th March 2005

Informatics and Mathematical Modelling

Technical University of Denmark

Preface

This thesis is the final requirement for obtaining the degree Master of Science in Engineering. It was written at the section for Computer Science and Engineering of the department of Informatics and Mathematical Modelling at the Technical University of Denmark. The project was carried out from September 1, 2004 to March 11, 2005, and was supervised by associate professor Christian D. Jensen.

We would like to thank Nete Kodahl and Torsten Lund Olesen for proof reading and our supervisor for his constructive criticism.

DTU, March 11 2005

____

Henrik Christensen

____

Jonas Høgh

Abstract

Traditional access control models which rely on a centralized reference monitor are not well suited for large-scale distributed systems. Cryptographic access control is a decentralized model, where access control is enforced solely based on possession of cryptographic keys. By including this access control scheme directly at the inter-process communication level, a distributed system can be created, where the confidentiality and integrity of all communication is built in by default, and where only authorized nodes are granted access to the system's assets.

This thesis therefore investigates the possibilities of incorporating the cryptographic access control model into the Remote Procedure Call (RPC) protocol. RPC is an inter-process communication paradigm that seeks to allow a program residing on one machine to call functions on another machine in a way similar to making a local function call. We design and implement a prototype RPC library based on the original Sun Microsystems RPC implementation. This includes extending the RPCgen code generation tool to be compatible with the new RPC library. We also look at alternatives to the port mapping system used by RPC to locate resources on a server.

Keywords

Cryptography, cryptographic access control, inter-process communication, remote procedure call, security

Resumé

Traditionelle modeller for adgangskontrol, der afhænger af en centraliseret reference-monitor, er ikke velegnede til store distribuerede systemer. Kryptografisk adgangskontrol er en decentraliseret model, hvor adgangskontrol håndhæves udelukkende udfra besiddelse af kryptografiske nøgler. Ved at inkludere denne form for adgangskontrol direkte på niveauet for inter-proces kommunikation, kan et distribueret system skabes, hvor konfidentialitet og integritet er indbygget som standard, og hvor adgang til systemets resurser kun gives til autoriserede klienter.

Dette eksamensprojekt udforsker derfor mulighederne for at inkorporere kryptografisk adgangskontrol i Remote Procedure Call (RPC) protokollen. RPC er et paradigme for inter-proces kommunikation, der tillader et program på én maskine at kalde en funktion på en anden maskine på en måde, der ligner et almindeligt lokalt funktionskald. Vi designer og implementerer en prototype af et RPC-bibliotek baseret på den oprindelige implementation fra Sun Microsystems. Herunder udvides RPCgen kodegenereringsværktøjet, således at det er kompatibelt med det nye bibliotek. Vi ser også på alternative metoder til at lokalisere resurser på en server, der kan erstatte den portmapper-mekanisme, der bruges i RPC.

Nøgleord

Kryptografi, kryptografisk adgangskontrol, inter-proces kommunikation, remote procedure call, sikkerhed

Contents

Chapter 1

Introduction

The arrival of almost ubiquitous internet access has made network security one of the most important disciplines in computer security. The ability to protect network-connected assets against unauthorized use has become critical to both corporations and individuals. Unfortunately, the access control mechanisms currently available in popular operating systems were not designed with large-scale networks in mind. Often, they rely on an access control matrix, i.e. a table listing all users and all assets in the system and the permissions of each user with regard to each asset. For this approach to be secure, all cooperating machines must agree on a way to consistently identify and authenticate the users. Synchronizing the information necessary to do this becomes impractical as the number of nodes grows.

An alternative approach to access control, which does not require such a global state, and does not rely on user identification, is cryptographic access control. Instead of maintaining an access control matrix, access to an asset is granted based on the user's possession of a cryptographic key associated with the asset. This is validated by requiring the user to encrypt his request for the asset with the key. Upon receiving a request, the system will attempt to decrypt it. The request will only be meaningful if it was encrypted with the correct key to begin with.

Since cryptographic access control has these advantageous properties in distributed systems, it seems intuitive that the most appropriate way to incorporate this access control scheme in a system, would be to add it to the inter-process communication layer. This will mean that all network services can be designed with access control taken into account, using a unified scheme across all applications. The use of encryption in the access control mechanism also has the benefit of making all communications unreadable to eavesdroppers.

In this report, we will propose a design of a version of the remote procedure call inter-process communication model using cryptographic access control. We will implement a prototype of the design, and evaluate its viability.

1.1 Overview

The remainder of this report is organized as follows: Chapter 2 covers various methods of inter-process communication, focusing on the remote procedure call model. Chapter 3 introduces the concept of security in computing, and describes techniques for securing communication. In chapter 4, we develop a model for integrating remote procedure call and cryptographic access control, which is elaborated into a system design presented in chapter 5. Our prototype implementation of this design is described in chapter 6, and is evaluated in chapter 7. We conclude the report by summarizing our findings in chapter 8.

1.2 Prerequisites

The reader is assumed to have knowledge of network protocols, particularly the internet protocol family. Familiarity with the UML notation is also expected. A basic understanding of UNIX-like operating systems is also helpful.

Chapter 2

Inter-process Communication (IPC)

As the name implies, inter-process communication covers any exchange of data between two or more processes, either within an operating system, or in a distributed system where the processes reside on multiple machines. Many methods exist for performing such communications in modern operating systems. In this section we will review some of the approaches, as well as the differences between them. The most common IPC mechanisms are based on message passing, shared memory, or remote procedure call (RPC). In particular we will focus on the latter model.

2.1 Message Passing

Informally, by a message passing model is understood any IPC model that is based on transferring a piece of data from a sender to a receiver. This definition of course covers a broad scope of systems, that differ in numerous ways. The communication may be either synchronous or asynchronous, i.e. the sender may wait for the receiver to successfully receive the data, or the sender may continue immediately without any such confirmation. Different access control schemes are also used - some systems are completely unrestricted whereas others use file-like permission bitmasks. Examples include named pipes, which are simple uni-directional FIFO-buffers, and the BSD socket abstraction, which is the basis of network communications using the TCP and UDP protocols, as well as UNIX Domain Sockets used for internal communication between processes in the same system. Higher level systems for communicating between nodes in a distributed system also exist, one such is the Message Passing Interface, or MPI.

2.2 Shared Memory

Generally, the term shared memory covers any mechanism for allowing multiple processes to share a virtual memory space. This approach has the advantage of being very fast, and allows for non-sequential access to the shared data. However, this comes at a price in programming complexity, since the lack of any built-in mediation of access to the data allows the programmer to create code that is prone to race conditions.

The implementation of shared memory used in most modern UNIX variants is based on AT&T's System V. It defines a set of system calls for creating and managing shared memory segments, and a semaphore mechanism for coordinating access in order to avoid the problems just mentioned. An access control scheme similar to standard UNIX file permissions is available for shared memory segments, enabling the programmer to assign read and write access to individual users and groups in the system.

Shared memory models also exist in distributed systems, but these are significantly more complex. A coherency protocol is needed in order to ensure that all nodes agree on the contents despite concurrent accesses. The performance gained from working directly in the same memory segment is of course also impossible to maintain if the contents have to be transported over a slow network link.

2.3 Remote Procedure Call (RPC)

Remote procedure call is a client/server infrastructure which enables a process to invoke functions residing on a remote system in a way that closely resembles a regular call to a local function. This is done by using client and server stubs, where the client stub works as a representative of the remote procedure. When an application wants to call a remote procedure, it will invoke the client stub, which will then call the server stub. The procedure is then executed by the server stub and the result is returned to the client stub, and further on from the client stub to the application. This is illustrated in figure 2.1. For an

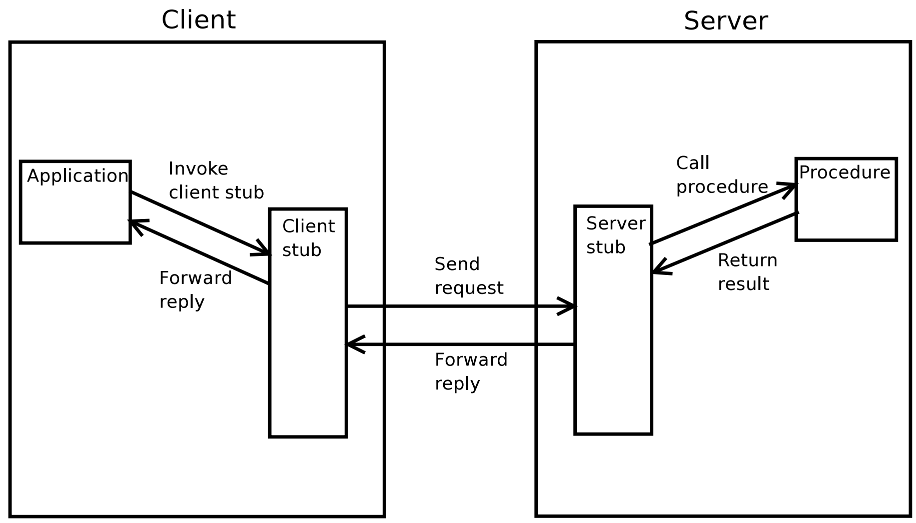

Figure 2.1: The remote procedure call model

RPC system to work the underlying system must provide a suitable low level transport protocol (such as TCP or UDP). The clients must be able to identify on which port they can communicate with the different remote procedures. This can either be hardcoded into the procedure or the system can use some form of naming mechanism. If the system should be able to communicate over different machine architecture, operating systems, and languages, there must exist some way of passing data from the clients native data representation to the servers and back again.

RPC systems differ from most other forms of IPC models by using function shipping instead of data shipping, that is, it operates with procedure calls instead of data transfers. Examples of RPC systems includes: Distributed Computing Environment (DCE) RPC and Open Network Computing (ONC) RPC. The latter is the one on which we will base our system.

2.4 Open Network Computing (ONC) RPC

ONC RPC originates from the RPC system developed by SUN Microsystems in the 1980's[1]. It allows client and server processes to communicate across different hardware and operating systems by using its own interface definition language, which is an extension of the eXternal Data Representation, XDR, language [1].

An RPC service can either run on a predefined port or obtain a new port on start up. RPC provides a service for RPC services to apply for a TCP/UDP port. Two programs that can be used for this are PORTMAPPER and RPCBIND [2]. Portmapper always runs on port 111, and when a new RPC service is started, it contacts the portmapper to obtain a port. When an application wants to execute an RPC service it contacts the portmapper, obtains the port associated with the specified RPC service and initiates communication. Figure 2.2 illustrates the RPC model.

In the figure, only one process is active at any time, this is only an example, since the RPC protocol makes no restriction on concurrency. RPC calls may be asynchronous, for example, the client process could perform other tasks while waiting for a reply from the server, or the server may create a new task for each incoming call, leaving it free to service other requests.

2.4.1 The RPC Interface Definition Language (RPC IDL)

RPC IDL is used to define programs, and to marshal the argument and result of a remote procedure call. RPC IDL uses XDR to do the marshalling of the data before it is transmitted over the network, and to convert the data back to the systems native data representation when a marshalled object is received. To define programs, RPC IDL has added two keywords to the XDR standard: *Program* and *Version*. These are used to define which procedures are available in each version of each program. For example, the following code is the definition of a program with one version, which contains one procedure:

```
program MESSAGEPROG {
    version MESSAGEVERS {
        int PRINTMESSAGE(string) = 1;
    } = 1;
} = 0x20000099;
```

[1] from here on when we mention RPC it will refer to ONC RPC unless otherwise stated

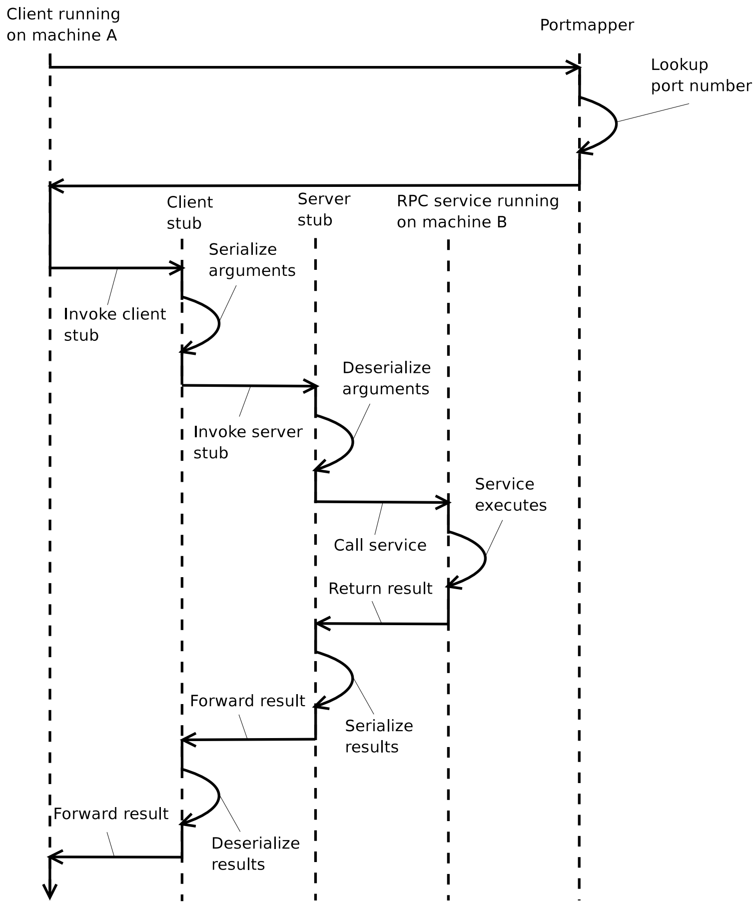

Figure 2.2: The remote procedure call model

Here a remote program with a single procedure, `PRINTMESSAGE`, is declared in version 1 of the program, `MESSAGEPROG`. The program, version, and procedure are each assigned a number, this triple can be used to uniquely identify the procedure. The programmer can freely choose the version and procedure numbers. The only restrictions are that they must be natural numbers, no two procedures can have the same number in one version, and no program must have two versions with the same number. In order to ensure that each program has a unique number these are administered by a central authority, which today is Sun Microsystems[2]. The program numbers are divided into groups of 0x20000000, as shown in table 2.1.

[2]Recent internet standard drafts propose that the authority is to be transferred to the IANA in the near future.

Program Numbers	Description
0x00000000 - 0x1fffffff	Defined by Sun
0x20000000 - 0x3fffffff	Defined by user
0x40000000 - 0x5fffffff	Transient
0x60000000 - 0xffffffff	Reserved

Table 2.1: Program number assignment

- The first group is the numbers administered by Sun Microsystems
- The second group can be used for programs, that are supposed to run on a specific site, but is primarily intended for debugging new programs
- The third group is reserved for programs that generate their numbers dynamically
- The rest of the groups are reserved for future use

To marshal the arguments and results from remote procedure calls the programmer needs to specify some XDR filters. For each procedure he must specify one filter for the argument and one for the result. For most primitive data types filter routines are supplied by the RPC library. If the procedure uses more complex data structures, the programmer can use the primitive routines in the RPC library to specify the filters, or he can use RPCgen to generate these filters for him.

The RPC IDL only supports procedures with one input parameter and one output parameter. However, this is only a minor restriction, since XDR allows for several parameters to be grouped into one data structure, which then can be used as the input parameter and similar for the output parameter. The transformation of the parameters from the systems native data structure into an XDR data structure and back again is known as serialization and deserialization, respectively.

2.4.2 RPCgen

RPCgen is a tool the RPC programmer can use to automatically generate the RPC network interface code. RPCgen takes a program interface definition written in RPC IDL (see sec. 2.4.1) and produces a client stub, a server stub, and routines that convert arguments and results into XDR and vice versa.

RPCgen can save a lot of development time, since the programmer(s) can focus on the main features of the application instead of using time writing and debugging the low-level network routines. The interface generated by RPCgen "hides" the network from the client and server applications. In order to make a remote call, the programmer writes a main client application that makes a local procedure call to the client stub created by RPCgen, which handles the marshalling and forwards the call. On the server side the remote procedure needs to be implemented and linked to the server stub.

RPCgen accepts an interface definition file written in RPC IDL and outputs the generated code in the C language. An interface definition should contain a program definition and possibly some definitions of data structures. If the programmer chooses to define one or more data structures, RPCgen will generate

XDR filters for these structures. It is possible to use data types not supported by the XDR library and which are not defined in the interface definition file. But in doing so you must provide the XDR filters to handle this data type.

The following code is an example interface definition file, which contains data structures declared in XDR.

```
/* dir.x: Remote directory listing protocol */
const MAXNAMELEN = 255; /*maximum length of directory entry*/
typedef string nametype<MAXNAMELEN>;  /* directory entry */
typedef struct namenode *namelist;    /* a link in listing */

/* A node in the directory listing */
struct namenode {
    nametype name;        /* name of directory entry */
    namelist next;        /* next entry */
};

/* The result of a READDIR operation. */
union readdir_res switch (int errno) {
case 0:
    namelist list;   /* no error: return directory listing */
default:
    void;        /* error occurred: nothing else to return */
};

/* The directory program definition */
program    DIRPROG {
        version DIRVERS {
            readdir_res READDIR(nametype) = 1;
        } = 1;
} = 0x20000076;
```

Running RPCgen on dir.x generates four output files:

dir.h The header file which contains #define statements for the program. This must be included in the server and client applications

dir_clnt.c The client stub. This contains the client routine `readdir_1`, which is used in the client application

dir_svc.c The server stub. From here the procedure `readdir_1_svc` is called, this procedure must be present in the server application

dir_xdr.c The XDR filters for marshalling the argument and return data

The programmer can use the code generated by RPCgen directly, or he can choose to alter it as he sees fit.

2.4.3 Programming in RPC

The RPC interface offers RPC programmers different levels of complexity and flexibility. One way of describing this is as a series of layers, e.g. as it is done in the documentation for RPC in Digital UNIX [4].

The highest layer is basically the level at which end users will use RPC. This contains no direct interaction with the functions in the RPC library. Here, the programmer takes advantage of the work of other programmers and calls local procedures that take care of calling the remote procedure and returns the result, without the programmer needing to worry about the network and which operating system the server is running.

The middle layer also enables the programmer to call remote procedures without worrying about creating sockets and which operating system the server is using. The programmer is able to call remote procedures by using the following three procedures:

registerrpc is used to register a procedure on the server. It binds a unique procedure number to the procedure given in the parameters. It takes 6 parameters, where the first three are the program number, the version number, and the procedure number. The fourth parameter is a pointer to the procedure which is to be registered. The last two parameters are XDR filters to decode the parameters to the RPC procedure and encode the results. Procedures registered with this function always use the portmapper, and cannot be defined to run on a specific port.

callrpc is used by the client to call a remote procedure. It takes 8 parameters, the first parameter is the hostname of the server. The next three parameters are the program, version, and procedure numbers. The next two parameters are an XDR filter and the argument for the remote procedure. If the procedure takes more than one argument these are encoded in an XDR structure, which is done by the filter. The final two parameters are another XDR filter for decoding the result and a pointer to where the result should be stored. *callrpc* will serialize the arguments, send the request to the server, wait for the reply, deserialize the result and return this.

svc_run is called by the server when all procedures have been registered. It causes the server to enter an infinite loop, during which it listens for incoming requests. Upon reception of a request, control is transferred to the procedure body. *svc_run* also decodes the arguments and encodes the result using the XDR filters specified when the procedure was registered. *svc_run* has no parameters.

The simplicity of the middle layer greatly reduces the flexibility and therefore renders this layer inappropriate for complex programming tasks. With the middle layer the programmer is unable to specify timeouts, use another transport protocol instead of UDP, perform authentication on the client or server side, or implement his own error handling.

The lowest layer contains functions the programmer can use to change the default values for the options mentioned above. Here the above call to *registerrpc* is replaced by three procedure calls:

svcudp_create creates a transport handle used by the server to keep information needed in the communication, and also contains functions to receive and reply to RPC messages. As parameter it takes a socket number. If the socket is bound to a port number, the same port number must be used, when the client calls *clntudp_create*. Alternatively, the parameter can be RPC_ANYSOCK, in which case the procedure creates a socket itself. If the programmer wishes to use TCP instead of UDP, he may use *svctcp_create*. On success, a pointer to the transport handle is returned, otherwise NULL is returned.

pmap_unset takes the program and version number as parameters. It destroys all mappings of program and version to a port number from the portmapper tables, so that the portmapper tables at all times has at most one entry for each version of each program.

svc_register takes a transport handle, a program and a version number, a dispatch function, and a protocol as parameters. It is used to register the program version with the portmapper, that is, it creates a mapping in the portmapper tables from the triple (program number, version number, protocol) to a port number. It also creates an entry in a list, which associates the triple (program number, version number, protocol) with the dispatch function, where protocol is the last parameter given to *svc_register* and can be either IPPROTO_TCP or IPPROTO_UDP. If protocol is zero no binding is performed.

Here the registration is on program level instead of procedure level, as it is in the middle layer. Based on the result of these three calls the programmer can specify his own error handling. The XDR filters are specified in the dispatch procedure instead of in the registration. In the dispatch procedure the functions `svc_getargs` and `svc_sendreply` are used to deserialize the arguments and serialize the result, respectively. Besides serializing the argument `svc_sendreply` also takes care of sending the result to the caller. When all the procedures have been registered `svc_run` is called just as in the middle layer.

On the client side the call to `callrpc` is replaced with calls to the following three functions:

clnt_create takes the hostname, program and version number, and transport protocol as parameters. Actually it is a wrapper function which will create a socket, set the port number in the socket data structure to 0, and call `clntudp_create` or `clnttcp_create` depending on which transport protocol is specified in the parameters.

clntudp_create creates a pointer to a client data structure. It takes the server address, the program and version number, a timeout value, and a pointer to a socket as parameters. The time specified in the timeout value indicates the timeout between tries. If the port number specified in the socket data structure is 0, the portmapper on the server is queried to get the port number used by the service. In `clnttcp_create` the timeout value is omitted, instead the size of the receive and send buffer must be passed as parameters.

clnt_call is a macro which will call either `clntudp_call` or `clnttcp_call` depending on which transport protocol was specified as a parameter to `clnt_create`.

clntudp_call is used to call the remote procedure. It takes the client pointer created by `clntudp_create`, the procedure number, an XDR filter function for serializing the argument, a pointer to the argument, an XDR function for deserializing the result returned by the remote procedure, a pointer to where the result will be stored, and a timeout as parameters. The timeout is the time in seconds for how long the function should wait for an answer. Should an answer not be received in this time period, another request may be sent to the server. `clntudp_call` will serialize the arguments, send the request, receive the reply, deserialize this, and return it.

clnt_destroy is a macro which works in the same way as `clnt_call`.

clntudp_destroy deallocates the space pointed to by the client handle. It will also close the socket associated with the client handle if this socket was opened by the RPC library. If the socket was opened by the user, it will remain open. This is done to make it possible to associate more than one client to a socket, and then destroy one client handle without closing the socket, which may still be used by other client handles.

Which layer of RPC to use depends on the amount of time one wishes to use, and how many details one needs to control. If the programmer needs to call a function on a remote server, and does not worry about how this is done, the easiest way is to use an interface which will wrap the serializing and remote procedure call into one single local call, thus the programmer only needs to specify the arguments and hostname of the server, even these may be wrapped into the interface. This is only possible if the RPC service is already configured and running on the server. If no such interface exists or the programmer needs to make his own interface the middle layer can be used. If the remote procedure takes multiple arguments, the programmer needs to write his own XDR filters to do the serializing, or he can use RPCgen to automatically generate these. This is done by using the *-c* option of RPCgen, which is also possible if the programmer uses the lowest layer of the RPC interface to control some of the more sophisticated details. The programmer can also choose to use RPCgen to implement the entire interface. RPCgen generates code which uses the lowest layer of RPC, thus making the programmer able to change details in the code if necessary. In practice the middle layer is seldom used, because programmers often need to change one or two of the default settings.

2.4.4 Secure RPC

The RPC standard allows inclusion of authentication information along with the transmitted calls and replies. It is possible to authenticate both the client and the server in this manner. The standard is open-ended in that various *flavors* of authentication are identified by unique integers, which are assigned by Sun Microsystems in the same manner as program numbers. The RPC standard itself defines two such flavors: `AUTH_NONE`, no authentication, and `AUTH_SYS` which allows the client to authenticate himself to the server using a UNIX (user ID, group ID) pair (this flavor is also known as `AUTH_UNIX` in some versions of RPC). The client and server must share user and group databases for this scheme to be useful. Furthermore, it is very easy for a malicious client to forge the credentials, e.g. by crafting packets with a zero user ID, which is normally used to identify the superuser.

Another authentication scheme called AUTH_DES or AUTH_DH, since it is based on the Data Encryption Standard algorithm[3], and the Diffie-Hellman key exchange protocol, was originally included in previous versions of the RPC standard in order to solve the problems inherent to UNIX authentication. It enabled the server and client to mutually authenticate one another. However, due to a bad choice of parameters in the implementation of Diffie-Hellman, and the fact that the key size used by DES had been widely regarded as insufficient due to the increased availability of faster and faster hardware, support for AUTH_DES was later removed from the standard. It would have been possible to solve the problems by changing the parameters and using another algorithm, but since a more generic and secure authentication scheme had been developed instead, it was decided to encourage users to move to this scheme instead of having two co-existing incompatible versions of AUTH_DES. The new standard, which moved beyond authentication by adding integrity and confidentiality[4], was based on the Generic Security Service Application Programming Interface (GSS-API), and is known as RPCSEC_GSS. Through access to this flexible API, the RPC programmer may implement almost any combination of algorithms, services, and mechanisms. Here, by services we understand the three main security properties provided by the flavor: Confidentiality, integrity, and authentication, or any subset of these. Mechanisms cover any higher-level security infrastructure that the system must integrate with, such as a Kerberos system or an X.509-based public key infrastructure. The downside to the flexibility of RPCSEC_GSS is of course that it leaves a lot of work to be done by the implementors of the underlying RPC programs. It is also a fairly complex system, a detailed description of which is beyond the scope of this project.

2.5 Summary

In this section, we have compared the three main methods of inter-process communication: Message passing, shared memory, and remote procedure call. The former two are data-shipping mechanisms. In message passing, data is actively transmitted from one process to another by some kind of channel, which may be either synchronous or asynchronous. In shared memory, on the other hand, the data is stored in a common area accessible to all the relevant processes. The main advantage of this approach is speed and the possibility to access data non-sequentially. RPC, in contrast, makes use of function shipping - it is designed to mimic regular procedure calls as closely as possible, by hiding the network communication specific code from the programmer.

To this end, RPC provides platform-independent serialization of data structures through use of the XDR standard. The RPCgen code generator facilitates RPC programming by partially automating generation of client and server stub code. Use of RPCgen is not mandatory, but RPC programming can be complex without it. The main requirements placed on the underlying system by RPC are the following: A suitable transport protocol must be available. In practice this is almost always TCP or UDP. A naming mechanism must also exist, enabling the clients to uniquely identify each program and its versions on each server. This is achieved by querying the portmapper for port numbers associated with

[3]See section 3.2 for an introduction to cryptographic terminology

[4]See section 3.1

(program number, version number) pairs. And the system must have a method to convert data, so it is able to communicate between different setups, e.g. different hardware or operating systems. RPC does this by serializing the data into a XDR data structure. The steps involved in performing an RPC call are illustrated in figure 2.2.

Chapter 3

Secure Communications

This section will first define security as the term is used in computing, and then give an overview of some of the technologies which are most commonly used to build secure communications. First, we will cover classical cryptography, then various schemes for naming network resources in a secure way. Finally, we will describe various models and mechanisms for access control.

3.1 Security

Traditionally, computer security is divided into three main aspects: Confidentiality, integrity, and availability. Only a system in which all three aspects are addressed is defined as a secure system. We will describe each aspect in more detail below.

Confidentiality means that only authorized parties have access to assets. It is also known as secrecy or privacy. It is a very simple and intuitive property in theory, but it can be difficult to implement. When an asset is available in a readable form, some kind of access control must be in place in order to determine which parties have access and which do not. Access control is described in more detail in section 3.4. Often it is combined with an authentication model in order to identify the user in a manner that cannot be forged by an unauthorized person. An alternative is to make the information unreadable altogether. This is the main purpose of cryptography, which is described in section 3.2. This method is widely employed when one wishes to communicate confidentially across an insecure network.

Integrity means that assets cannot be modified by unauthorized parties, or modified in unauthorized ways. It can be enforced through many of the same mechanisms as confidentiality. However, increased care must be taken in situations where an outside modification of an asset may go unnoticed. For example, even when a packet transmitted across a network is unreadable to an unauthorized person because of encryption, he may be able to compromise the integrity of the data by changing it in transit. Cryptography also offers a solution to this problem in the form of cryptographic hash functions, message authentication codes, and digital signatures.

Availability ensures that the system is accessible to the authorized parties at the desired times. This means that the system must be as responsive as possible in the face of high utilization levels, as well as attempts to overload it through large amounts of unauthorized requests. The system must also be robust with respect to handling malformed requests and other erroneous conditions. It is very difficult to strictly define guidelines for availability, since it is inherently subjective how long a user should be allowed to wait, how much downtime is acceptable, and so on. The concept of availability is unfortunately not nearly as well understood by security researchers as the two other aspects of security, even though many agree that it is becoming increasingly important.

3.2 Cryptography

Cryptography is the practice and study of encoding data so that it can only be decoded by individuals in possession of secret knowledge. The process of encoding is known as *encryption*, the reverse process as *decryption*. A message in its normal, readable form, is *plaintext*, in encrypted form *ciphertext*. A mathematical function specifying how to encrypt or decrypt is called a *cryptographic algorithm* or *cipher*. In modern cryptography, the cipher is usually made publicly available, but is dependent on a user-specific parameter, called the *key*, in addition to the data. This makes it possible to standardize on well-known, peer-reviewed ciphers, instead of relying on customized secret algorithms. A cipher together with the sets of all possible keys, plaintexts, and ciphertexts is known as a *cryptosystem*.

Cryptanalysis is the study of attacks against cryptosystems. Any system can be successfully compromised by an attacker who has the resources to enumerate all possible keys, a so-called brute force attack, but such attacks can be made prohibitively expensive by choosing a sufficient key size. A good cryptosystem ensures that more sophisticated attacks are not significantly more efficient than brute force.

3.2.1 Symmetric Cryptography

In a symmetric cipher, the same key is used for both encryption and decryption. When transmitting a message between two parties, it is therefore necessary to agree on a common key, which is kept secret to anyone else. For this reason, keys in symmetric systems are often called *secret keys*.

Symmetric ciphers are mainly classified into two groups: Stream ciphers and block ciphers. The former treat the plaintext as a continuous stream of bits, which are encrypted one by one, based on some internal state of the algorithm. A very simple example of a stream cipher is initializing a pseudo-random number generator with the key as the seed, and xor the output with the plaintext. Block ciphers, on the other hand, treat the plaintext as a sequence of fixed-size blocks, e.g. of 64 bits or more. In its simplest form, the cipher operates on each such block independently and statelessly. A disadvantage of this approach is that any given plaintext block will always encrypt to the same ciphertext block, no matter what context it appears in. To prevent this problem more advanced modes of operation have been devised for block ciphers. One such mode is *Cipher Block Chaining (CBC)*, in which each block of plaintext is xor'ed with the previous block of ciphertext before it is encrypted. For the first block, a randomly generated block of data, called the *initialization vector* is used. Other examples of modes are *Cipher-Feedback (CFB)* and *Output-Feedback (OFB)*, which effectively transform a block cipher into a stream cipher by using a shift register. The register initially contains an *initialization vector*, and is then incrementally filled from one end with small (e.g. 1 byte) segments of ciphertext (in CFB) or the key (in OFB). The key is generated by encrypting the contents of the shift register, and taking e.g. the leftmost 1 byte. Again the main purpose of introducing these modes is to conceal plaintext patterns. CFB and OFB can also be a convenient way of yielding the benefits of stream ciphers, when these are needed, e.g. the ability to encrypt a single character and transmit it independently in a secure terminal application. When the naive approach of block-by-block encryption is compared to one of these alternative modes, it is known as *electronic cookbook mode*, or ECB.

3.2.2 Asymmetric Cryptography

Asymmetric cryptographic, or public-key cryptography, as it is also known, was first described by Whitfield Diffie and Martin Hellman in 1976. In an asymmetric system, two distinct keys are used for encrypting and decrypting a message. The two keys are mathematically related, but it is not feasible to obtain one from knowledge of the other. This makes it possible for a person to publicly distribute his encryption key, while keeping his decryption key secret, thus enabling anyone with access to the encryption key to create a message which only he can read, without any need for an exchange of keys. For this reason, the encryption and decryption keys are often called *public* and *private* keys, respectively. However, it is important to note, that even though the owner of a public key has no need to keep it secret, an imposter may distribute *another* key in his name, enabling the imposter to read the encrypted messages rather than the intended recipient. For this reason, public keys must still be obtained from a trusted source over a secure channel. This is one of the major practical limitations of asymmetric cryptosystems.

Mathematically, public-key systems rely on trapdoor one-way functions. A one-way function is a function for which it is easy to calculate function values, but no polynomial-time algorithm exists for calculating the inverse function. A trapdoor one-way function has the additional property, that the inverse function is easy to calculate given a piece of additional information. This information is the private key. The existence of one-way functions is unproven, and proving it would imply solving the classical $P = NP$ problem. Actual public-key systems are therefore based on functions conjectured to be trapdoor one-way functions. Probably the most well-known example is the RSA[1] system, which is based on the factorization problem. The trapdoor is constructed based on certain number-theoretical properties of modular arithmetic, and is easily calculated since the prime factors are known. Other examples of conjectured trapdoor one-way functions which have been used in cryptosystems are elliptic curves and discrete logarithms.

In general, asymmetric cryptosystems are significantly slower than symmetric ones, and the key lengths required to achieve comparable levels of security are much larger. For these reasons large amounts of data are rarely encrypted with a pure asymmetric algorithm. Rather, the asymmetric algorithm is used

[1] Named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman

as a means of transporting the symmetric key, which is then used to encrypt the data. This combination of the two classes of cryptosystems is called a *mixed-mode cryptosystem*

3.2.3 Cryptographic Hash Functions

A *hash function* is a function that takes a variable-size input (called the *pre-image*), and generates a fixed-size output (called the *hash value).* Generally, hash functions are used in many areas of computer science where a fingerprint of data is required. Obviously, the fact that hash functions map an infinite domain to a finite range, will result in *collisions*, i.e. the functions are not one-to-one. Often this property is undesirable, e.g. it will cause a degradation of performance of data structures based on hash functions such as hash tables. In spite of this, by simply using a good hash function that distributes the possible inputs evenly over the possible outputs, good average performance can be achieved.

In cryptographic contexts, it is often desirable to fingerprint data. Applications include message checksums that protect the integrity of the message, and performance improvements when an expensive operation can be performed on a fingerprint rather than an entire message. However, more serious problems than the ones mentioned above arise from hash collisions when we use hash functions for these purposes: If an attacker is able to manipulate such a system by finding collisions, he might be able to substitute one message for another with the same fingerprint. For this reason, only hash functions that satisfy certain properties are classified as cryptographic hash functions. The following properties are generally considered desirable:

Pre-image resistant: Given a hash value, it must be hard to find any corresponding pre-image

Second pre-image resistant: Given a pre-image, it must be difficult to find a different pre-image with the same hash value

Collision resistant: It must be difficult to find any two pre-images that share the same hash value

This is not an exhaustive list, and none of the requirements are formal. As with other cryptographic primitives, hash functions are not provable secure.

When it is desirable that a fingerprint can only be verified by the intended recipient, one may add a key to a cryptographic hash function, e.g. by combining it with a symmetric block cipher. Such a construction is known as a *Message Authentication Code* or MAC.

3.3 Naming Schemes

As mentioned in section 2, a method of naming servers, programs and procedures is a prerequisite for an RPC mechanism. We will therefore look at alternative naming methods in this section. Various technologies exist for creating a binding between e.g. a network address and a cryptographic key. These help prevent address spoofing attacks, as long as the key is not compromised. We also look at the method for keeping track of documents used in the Freenet peer-to-peer network, which is a way of generating a fingerprint based on a descriptive string and a cryptographic key.

3.3.1 Cryptographically Generated Addresses

Cryptographically Generated Addresses (CGA) originates from Greg O'Shea and Michael Roe's Child-proof Authentication for MIPv6 (CAM) protocol [5]. The basic idea is, that a public key is bound to the 64 bit interface identifier of an IPv6 address. This is done by computing a one-way hash function from the public key and using this as the interface identifier. The binding between the IPv6 address and the public key can then be verified by recomputing the hash function, and the host can claim ownership of the IPv6 address by signing messages with the corresponding private key.

The host identifier is defined as:

$$HostID = Hash_{62}(Public\_key)$$

The interface identifier can also be created from a hash chain:

$$H_N = Hash_{160}(Public\_key \| random)$$

$$H_i = Hash_{160}(Public\_key \| H_{i+1})$$

$$HostID = Hash_{62}(H_0)$$

This enables the host to claim ownership of the address without using a signature. If a collision of addresses occur both parts reveal their H_1. Since this is 160 bits long the chance of a new collision is very slim. If the two values of H_1 collide the reason most probably is that one of the them learned the value from the other. Then they both reveal H_2 to verify the ownership of the IPv6 address. If it is nessecary to reveal hash values beyond H_2 the host can be certain that an attack is occuring.

The Internet Engineering Task Force (IETF) has proposed algorithms for CGA generation, verification, and signature [6]. To create a CGA the algorithm needs the public key, a 64-bit subnet prefix, and a security parameter, *Sec*. *Sec* is an unsigned integer and is encoded in the three leftmost bits of the CGA. *Sec* determines the CGA's strength against brute force attacks.

Secure Neighbor Discovery (SEND) Protocol

An IPv6 host uses Neighbor Discovery to discover routers and other hosts on the link, and to maintain reachability info.

The Internet Engineering Task Force (IETF) has for some time tried to secure neighbor discovery in IPv6 without using IPsec, since no methods for using IPsec without manual keying exist today, and one of the goals of neighbor discovery is to achieve security without manual configuration. As described in [7] and [8] CGA can be used to secure neighbor discovery. If the neighbor discovery messages are signed by the key corresponding to the CGA which the message came from, the receiver can be certain that neighbor advertisements and neighbor detection messages are authentic.

3.3.2 Address Based Keys

Address Based Keys (ABK) is a cryptographic technique where a node's public and private keys are generated from its IPv6 address. ABK is based on a technique called identity-based cryptosystems, which was first introduced by Shamir [10] in 1984. Shamir wanted to remove the need for key directories and certificates from public key cryptosystems by using the receivers identity as the public key. Schemes for identity-based signatures and key agreement protocols quickly followed, but it was not until 2001 that a secure and efficient identity-based encryption scheme was proposed by Boneh and Franklin [11].

Identity Based Cryptosystems

An Identity Based Encryption (IBE) scheme is a public key encryption scheme where the public key can be any string, which uniquely identifies the user and is readily available to the other part. This could be an email address, a telephone number, a social security number, a net address, or any combination of these.

A user chooses a public key and sends this to a trusted third part, which is called Identity-based Private Key Generator (IPKG) and then receives his private key over a secure channel. The IPKG generates the private key from the public key, some publicly known parameters and a master key, which is only known to the IPKG. The publicly known parameters are generated from some chosen constants and the secret master key. Besides being used in the generation of the private keys, these are used by the users to perform cryptographic operations.

Since the public key can be any string, a user can choose a string which identifies the person he wants to send an encrypted message to, i.e. Bob wants to send an encrypted email to Alice, he then encrypts the message with Alice's email address and sends the mail to Alice. Alice will then send her email address to the IPKG and receive the appropriate private key. Alice is then able to read the message.

In a public key encryption scheme there is an expiration date included in the certificates. This can be incorporated into IBE by including the date in the public key, i.e. one system could use the email address concatenated with the current month and year as public keys. This will ensure that the users change their keys every month.

Key escrow is built into the system, since the IPKG can recreate a users private key if it knows the master key and public parameters which was used to create the private key. The master key may be distributed among several IPKGs by using threshold cryptography, to ensure that one IPKG cannot impersonate or snoop messages from a user, for which it generated a private key.

Using ABK to Secure IPv6 Neighbor Discovery

ABKs use part of an IPv6 address as the public key. A host would use its 64 bit interface identifier as public key, and a router would use its 64 bit subnet prefix. It has been proposed that ABKs are used to secure IPv6 Neighbor and Router Discovery[12], so that router advertisements are signed by the router and neighbor advertisements are signed by the host. This would remove the key management problems in IPSec.

Another proposal is to use ABK for router discovery and cryptographically generated addresses (CGA) for neighbor discovery[7]. The advantage of using CGA to do neighbor discovery is that it does not require any trusted third part.

3.3.3 Freenet

Freenet is a distributed data storage system, where the identity of the submitter and the reader of data is protected. It has a peer-to-peer architecture, where every user provides a node, where he provides some data storage for the system.

When a user commits or requests some data, the data is sent through a chain of nodes. Each node can only see its immediate neighbors, and has no way of knowing whether the node, from which it receives a request or insert message, is the originator of this message or whether it is only forwarding the message from another node. This is to ensure that a malicious node cannot obtain information about the submitter of the request or the holder of data. Each message has a *hops-to-live* value which indicates how long the request will travel before it returns if the file is not located.

Each file is assigned a global unique identifier (GUID) key. These are calculated using SHA-1. Each file is submitted together with its GUID key, and these are used to locate files. Each node has a routing table which associates GUID keys with nodes. Each request contains the GUID key for the wanted file. When a node receives a request, it first checks its own datastore for the file. If it is not found, it finds the key in its routing table which most resembles the requested key, and forwards the request to the corresponding node. If the node locates the file in its own datastore, it returns the file to the node, from which it received the request, which then passes it on to the next node in the chain and so on, until the file reaches the origin of the request. In this chain each node inserts the GUID key and the holder of data into its routing table, and may choose to alter the reply message to point to itself as the data holder. A node may also choose to store the file in its own datastore. To insert a file an insert message containing the GUID key is sent. When an insert message is received the node checks if it already has the key. If this is the case, it returns an error message together with the file already associated with the key, just like it would do, if it received a request for the key. This means that malicious users, who try to insert corrupted files under existing GUID keys, only help spreading the original file. If the GUID key is not found before *hops-to-live* reaches zero an *all-clear* message is sent back to the submitter, who then sends the file along the chain, just as if it was a reply to a request.

There are two main types of GUID keys in Freenet; *Content-Hash Keys*, CHK, and *Signed-Subspace Keys*, SSK. A CHK is generated by hashing the content of the file, and is used to store files in Freenet, while SSKs are used to store pointers to CHKs and to ensure the authenticity of the file's submitter. An SSK allows a user to set up a personal namespace, that only the user can write to but everybody can read from. An SSK is created by generating a random public/private key pair. New files are then added to the namespace by choosing a descriptive text string for the file, calculating the hash value of this string and concatenating it with the hash value of the private key, and hashing the outcome of this to produce the file key. These two kinds of keys are often used in unison to provide versionable files. To update a file the owner first inserts the file under its CHK and then updates the SSK to point to this new version.

This means that the newest version of the file will always be available by the SSK, while a user can gain access to older versions by their CHKs.

Freenet supports a third kind of key called *Keyword-Signed Key*. The user chooses a short descriptive text string which is used to deterministically generate a public/private key pair. The public key is hashed to produce the file key. KSKs are rarely used, since they form a flat global namespace, so nothing prevents two users from choosing the same key.

3.4 Access Control

Access control is a method of restricting access to objects to authorized subjects. Objects are defined as the resources or information in a system. For example, an object could be a data file, a printer or a process. Subjects are the active entities that can cause information to flow between objects. For example, this could be a process or a user. In this section we will describe different kinds of access control models and some implementations of access control.

3.4.1 Access Control Models

Access control has traditionally been divided into two models; discretionary access control and mandatory access control, but in the last couple of decades a new model has evolved: Role based access control. We will describe these models here with emphasis on role based access control.

Discretionary Access Control (DAC)

In DAC, access is based on the identity of subjects and/or groups to which they belong. It is discretionary in the sense that a subject may pass access rights on to other subjects. In some systems it is only specific subjects who are able to pass on access rights to an object (e.g. the owner of this object).

In DAC, access rights can either be assigned to an object (like access control lists) or a subject (like capabilities). That is, an object can have a set of permissions associated, which grant access rights to subjects. These permissions specify which subjects have access and what kind of operations the subject can perform on the object (e.g. read, write, execute). Alternatively, each subject can have a set of permissions, which gives access to different objects.

Mandatory Access Control (MAC)

MAC is based on security labels: Each object and subject is assigned a label. When a user tries to access data the MAC policy compares the label of the user with the label of the data. The user is denied access unless the MAC security checks are passed. Users are unable to grant or remove access to data, hence the name “mandatory”. Only a core of security administrators are able to set and alter access security labels. Labels are often assigned descriptive names such as *top secret* or *confidential*.

Labels are assigned to objects based on the sensitivity of the information they possess. Each subject is assigned clearance that sets the upper and lower bound of a set of security labels, which the subject can access, these are set according to the job responsibility of the subject.

MAC enforces write up/read down rules, that is, users are only able to read data on their own or a lower security level, and it is not possibly to write data into the files of a lower security level. This is to ensure that *top secret* data is not written to a file with a label of *secret*. It is only possible to create files on the security levels which are accessible to the user.

Role Based Access Control (RBAC)

The management of permission and restriction in multi user systems can become quite extensive when the number of users increase. Role Based Access Control (RBAC) is one way of reducing the cost of security management. In many organizations it is natural that each employees security level is specified by that persons job function. E.g. in a hospital system, the "doctor" role would be granted access to write prescriptions, whereas the "nurse" role would not.

With roles the access control policy is more stable, since the activities and functions of an organization change less frequently than those of a single user. Therefore, the access control policy for a single role is seldom changed, while every time an employee changes job function the security administrator only needs to assign the employee to the new role and remove him from the old one.

RBAC has its roots in the concept of user groups as seen in UNIX and other operating systems, where access to files and directories can be given to a group of users. Here an owner of a sub-tree in the filesystem can give any group permissions to this subtree. So permissions are associated with each file and directory, and to determine all the permissions of a specific group one has to traverse the entire filesystem tree. In RBAC each role is a set of users and a set of permissions, and the role associates these two sets with each other. This describes one characteristic of RBAC: It shall be approximately equally easy to determine the members of a role and permissions assigned to a role.

Throughout the last decade many different variants of RBAC have been described. In the proposed standard from NIST [13] Core RBAC is the fundamental aspects which are required to achieve a RBAC system. The fundamentals in RBAC are the relationships between users, roles, and permissions: Users are members of roles, permissions are assigned to roles, and users obtain permissions by belonging to roles. The two relationships user-role and role-permission are many-to-many, that is, each user can belong to several roles and each role can have multiple users, and similar with permissions. Core RBAC also requires that it is possible to determine the members of a specific role, and the roles which a specific user is member of. Core RBAC can be extended to include other aspects such as hierarchical roles or separation of duty, both are described in [13].

Core RBAC can be implemented both as MAC and DAC, or it can coexist with an implementation of either one.

3.4.2 Access Control Mechanisms

We will now continue by looking at some of the implementations that have been used to realize the models described above. In particular, the Cryptographic Access Control paradigm will be described.

Access Control Lists (ACLs)

A common way to control permissions is by using ACLs. They are used in many of the most popular operating systems to control access to the filesystem, and in firewalls to filter IP addresses.

In a file system or similar systems each object has an associated ACL. These ACLs determine which access rights the users of the system have. Since the management of a system with an extensive amount of objects is overwhelming, ACLs are implemented according to the DAC model. In this way, the management of the ACL associated to a specific object is the responsibility of the owner of this object.

For each user or group which is permitted or denied access to an object, an entry is made into the corresponding ACL. These are called Access Control Entries, ACE. When a user or process tries to access an object, the system runs through all the ACEs for this object until one of them states, that the subject is either permitted or denied access. If no ACE regarding this subject is found access is denied. Usual negative ACE's, those that deny access, takes precedence over positive ACEs, those that permit access. That is if a group A needs access to a particular file but a subgroup of A called B does not, the first ACE would state that members of B are denied access, and the second would state that members of A is granted access.

Capabilities

With ACLs we have a list for each object stating which access rights each subject has. For capabilities each subject has a set of access rights to objects. Capabilities can be seen as tickets, since a subject "shows" that it has the correct access rights, when it tries to access an object. Basically a capability is a pair (x,r) where x is an object, and r is the access rights this capability grants to that particular object. Since no info about the subject is given in the capability, these must be unforgeable. Otherwise a user would be able to grant access to everyone. If it is practical to allow users to pass on capabilities, a special capability which allows users to grant access rights to other subject can be granted to these users.

The set of capabilities owned by a subject defines a domain. This domain is the collection of objects for which the subject has access rights. When the subject calls a procedure, this procedure also has its own domain and it may have access to some objects not accessible by the calling subject, the subject may also pass some of its own access rights along to the procedure.

Cryptographic Access Control (CAC)

The growing need to share and access data across large networks, where the user has to rely on third part elements, over which they have no influence, increases the need for effective security systems.

Cryptographic Access Control is one solution to this problem. The basic idea is to encrypt all data, and only users who knows the right cryptographic keys are able to read or write the data. This allows users to send data across an untrusted network or store it on a public server, without worrying about the confidentiality of the data. If a distinction between read and write access is required, one can use an asymmetric cryptosystem, where the encryption key is used to grant write access and the decryption key is used to grant read access.

Since only the clients know the encryption and decryption keys, the confidentiality of the data is preserved even though the data is sent across an untrusted network or the server on which it is stored is compromised. So even though a malicious person can easily gain access to the the data, he is not able to read or write the data, because he does not have the right keys. Also, by using digital signatures, the integrity of the data can be ensured.

CAC is suitable to be used in a large distributed network, where other forms of access control are too centralized to keep an overview of the current access control state in a distributed system. Most access control mechanics rely on a reference monitor in order to grant access. Since the actual verification of access needs to be done on the server that manages the desired object, this is too centralized to work in a distributed system. For a distributed reference monitor to work, it needs to have a synchronization system which at all times can ensure, that it has a consistent overview of the access control state, this is theoretically impossible, because one cannot ensure that failures will not occur in the system. This is true for ACL and capabilities, where a reference monitor is needed to check whether or not a client, who tries to obtain access to an object is allowed so according to that objects ACL, and likewise a reference monitor is needed to check a clients capabilities. In CAC a client is always allowed access, but is only able to read or write the data if he has the right keys.

CAC in a Distributed File System

In [14], Anthony Harrington and Christian Jensen described and implemented a distributed file system which uses cryptographic access control. They propose to use a mixture of symmetric and asymmetric cryptography to ensure confidentiality and integrity of the data, and a log system to preserve the availability of the data. All data is encrypted with the symmetric key, which is distributed to all users of the system. The introduction of symmetric cryptography speeds up the process of encrypting and decrypting the files, since it is several orders of magnitude faster than asymmetric cryptography. A public/private key pair is used to cryptographically sign all files in order to preserve the integrity of the files and distinguish between read and write access. A user who needs read access is given the symmetric key and the decryption key, and he is then able to ensure the integrity by verifying the digital signature. A user who needs write access is given the symmetric key and the encryption key.

The server is also given the decryption key in order to verify that a user, who wants to write some modification to a file, has the needed rights to do so. If the digital signature cannot be verified with the decryption key, the server dismisses the write request from the user.

CAC in Peer-to-Peer Networks

Another application of cryptographic access control was developed by Søren Hjarlvig and Jesper Kampfeldt in [15]. By associating key triples (a symmetric key and a matching pair of asymmetric keys) to the files and users in a Peer-to-Peer filesharing network, it is possible to store files in the network that are only readable and writable to a selected group of users. The integrity of the files is also protected, and a version history is created to keep track of changes to a particular file. Group management is possible by distributing key rings between the users.

3.5 Summary

In this section, we have looked at some of the technologies that will be needed to build a secure version of RPC based on Cryptographic Access Control.

Cryptographic algorithms can be used to protect the confidentiality of messages. Symmetric algorithms require a pair of communicating users to share a secret key, whereas asymmetric systems allow a user to distribute his public key to anyone, enabling them to construct messages only readable by the owner of the corresponding private key. Cryptographic hash functions are useful for securely fingerprinting data, and protecting data integrity.

We have also looked at two schemes for binding a cryptographic key to a network address. In the Freenet peer-to-peer network, hashing of descriptive names is used to identify documents.

Access control can be used to restrict access to objects. Several models are used to describe access control schemes: In DAC, permissions are assigned to each object. These can be modified by the owner of the object, which can be problematic. In MAC, only security administrators may modify access control information, which is in the form of labels, a range of ordered security levels, e.g. going from top secret to unclassified. A subject is only able to read objects with a lower (or identical) label, and write objects with a higher (or identical) label. In RBAC, subjects are assigned membership to a set of roles, which may be hierarchical. Permissions are assigned to roles, rather than directly to subjects.

Access control is most often implemented in operating systems through access control lists. Each list contains the permissions for a given object. Access to modifying the list is often left to the users themselves, making ACL's an implementation of the DAC model. Capabilities are instead centered around the set of rights that a given subject has. The individual capability object is a form of secure ticket, which the user cannot manipulate. Possession of the capability is therefore proof of the permissions listed in the capability. Cryptographic access control is a method where a permission is proven by possession of a cryptographic key. All messages in the system are encrypted, and requests are only accepted if they decrypt meaningfully, proving that the subject possesses the proper key.

Chapter 4

Secure RPC with Cryptographic Access Control

In this section we will describe the problems which we will address in this report. We will build a model of the system and discuss the different methods used in this model. Based on the model we will formulate a requirement description.

4.1 Background

With RPC it is easy to program and set up a service that can be accessed remotely, and if the programmer is not careful, this will give malicious persons unwanted access to different resources. This may either compromise sensitive data or in other ways be a nuisance for the users of the system.

A server which runs numerous services will almost certainly have one or more security holes. Even if some kind of protection from malicious persons is incorporated into these services, like encryption of data, access control or similar, the implementation of these may contain security holes. Maybe the code that handles the access control gives the possibility for a DoS attack or lacks sufficient checks for buffer overflow, and thereby has the opposite effect of what was intended. Also, if the security is handled separately by each service, it potentially leaves different security risks for each service. We wish to incorporate the security into the RPC library, so the amount of code that needs to be maintained is minimal, and the number of potential security risks is reduced. We will rely on two ways to ensure that the remote procedure is protected from malicious persons: Access control and dynamic assignments of port numbers.

We wish to use dynamic assignment of port numbers in a way, so that only the server and the authorized clients will know on which port a particular service is running. This is done to make it harder for an attacker to know which services a given server is running. As it is today, an RPC service can either run on a predefined port assigned by Sun Microsystems or obtain a new port on startup, as described in section 2.3. But the port number assigned by the portmapper is not secret, since anybody can ask the portmapper on which port the service is running, and therefore not a feasible solution to our requirement. We will design a solution based on the naming schemes described in section 3.3. Combined with access control this will ensure that even though a malicious person discovers that a server has a service running on a particular port, he is unable to determine which service this is.

4.2 Model

Here we will specify a model of the solution we wish to implement. We will discuss the different methods described in section 3, and how we will use these in our solution. During the description and analysis of the model we will give different requirements to the system. In the end we will gather these in a requirement specification.

4.2.1 Access Control

An administrator may need to manage a service remotely, without allowing regular users access to the procedures, with which this is done. Maybe a service even differentiates between advanced and normal users, so that the advanced user has access to some procedures that a normal users is not allowed to call. E.g. on a print server an administrator needs to manage the setup and the printers connected, an advanced user can add and delete jobs in a print queue, while a normal user is only allowed to add new jobs. As described in the example, the access rights of each user will be based on which user group they belong to. Furthermore, we determined in section 3.4.2 that access control methods based on a reference monitor are unsuitable in distributed systems, so we will base our access control on encryption. This means that the client needs to encrypt the messages before they are sent to the server, which will then decrypt them in order to verify that the request is legitimate. If the message is not on a form which is readable by the server after the decryption, the server will drop the message. This means that if a client sends a message encrypted with a wrong key, it will never receive an answer. This is done both to reduce the amount of resources used on false requests in order to make a DoS attack less feasible, and to increase the difficulty of determining the port number on which a service is running.

As mentioned we wish to use dynamic assignment of port numbers [1], so only authorized users know on which port a particular service is running. This also entails that a user cannot call procedures which are not accessible for the role he is currently assigned to. This means that each service must listen on as many port numbers, as it has roles, and upon reception of a remote procedure call verify that it is allowed to call the particular procedure on the port the request was received on.

The implementation of this access control must mainly reside in the RPC library, so we can ensure that RPC programs, that use our version of RPC, always use access control. Furthermore, it needs to be designed in a way, so that it is easy for the RPC programmer to assign procedures to the different roles.

[1] see section 4.2.2 for more details

4.2.2 Port Number Generation

We have determined that a service needs to listen on a port number for each role, these port numbers must be calculated from a shared secret, which is only known by the server and the members of a particular role. We also wants to give the RPC programmer the possibility to set up the service, so the port numbers change at a regular interval, to reduce the damage if a malicious person acquires the port number which is used by a service.

Changing the port number associated with a service frequently, will lessen the risk that a malicious person is able to compromise the service, but recalculating the port numbers will incur some cost. Choosing an interval for port reassignment is therefore a trade off between the cost of changing port number and the security level. The cost of changing port number should be as small as possible, and it must not prevent a user from starting a new conversation with a service or interrupt an ongoing conversation. Also, to increase the user-friendliness it should be transparent to the user, when the service changes port number.

In section 3.3 we mentioned cryptographically generated addresses and address based keys, at first we focused on these methods to generate the port numbers. This included extending the length of a port number to 128 bits, since these two method is based on IPv6, where an IP address is 128 bit. This much greater number of ports also greatly increases the cost of a port scan attack.[2] Thereby, we almost eliminate the vulnerability to port scans, one of the most popular ways of discovering which services a particular server is running. But using one of these methods would mean that the keys used in the cryptographic access control would be associated with a specific port number, and it would therefore be necessary to distribute new keys every time the port number is changed. Even though it is a good idea to generate new keys from time to time to ensure integrity, it is not necessary to do it as often as we would like to change port number.

Instead we use a technique similar to the way keys are created in Freenet, see section 3.3.3, to generate the port numbers. A descriptive string is created for each role in each program version. This should be done in a way, so that both the client and server are able to generate this string. The string is then concatenated with the hash value of the servers public key for the particular role and the result is hashed again to yield the port number. It is important that only members of a particular role are able to generate the descriptive string associated with this role, in order to make it infeasible to guess which port number is used by a role, if the public key for this role is obtained. Since the range of the hash function is limited, hash collision may occur, causing two procedures to listen on the same port, which means that one of these services is not allowed to start. Here we have another advantage of using 128 bit port numbers, the possibility that two descriptive strings hashes to the same value is less likely with 128 bit instead of 16. To ensure that two services do not try to run on the same port number, we need to do a check for the availability of the generated port number. If the port number is already taken, we need to have some predefined mechanism the client and server can use to generate another port.

[2] See section 4.2.4 for a more specific analysis of the security aspect of 128 bit ports.

4.2.3 Integrity and Confidentiality

To further heighten the security of the system, we need to introduce integrity and confidentiality to the conversations between client and server. Since we use CAC to verify access rights, the packets from the client to the server is already encrypted. All users in a role has access to the corresponding key, so to prevent these from decrypting the message sent from another member, the crypto system used in our CAC must be asymmetric. This way all the clients get the public key and only the server knows the private key, so no one else but the server can decrypt the requests encrypted with the public key. However, the private key cannot be used to encrypt the replies, because all members of a role knows the public key, and can therefore decrypt the reply. Besides this it is resource consuming to encrypt all messages with an asymmetric cipher. Instead we choose a mixture of asymmetric and symmetric cipher to increase performance and confidentiality: To initiate a conversation a client sends a message containing an RPC request to the server encrypted with the servers public key, the server will then know, that the client is allowed to call the particular procedure defined in the request. Along with the request, the client also sends a symmetric session key to the server, which will be used to encrypt the reply from the server. Each session key is only used once; they are discarded at the end of each conversation.

To provide integrity we add a fingerprint to the messages. This is done using hash functions as described in section 3.2.3. When a message is received either by the server or by the client, it will be decrypted and a check will be made to ensure that it has not been tampered with. This still leaves the issue of replay-attacks, where an attacker resends a legitimate message he has snooped from the network. To prevent this from happening we also attach a nonce to the first message sent by the client. The server will record this nonce when it receives the message, enabling it to detect if the same message is later replayed by an attacker. The reply from the server does not need a nonce, since the client will only listen for one message from the server and expects this message to be encrypted with the session key, and since the session key is only used once, it is not possible for the attacker to resend these replies.

To restrict access to current members of a role, it should be possible for the system administrator to change the role keys and the way in which the port numbers is calculated, so that it is not possible for an old member of a role to gain access to the procedures assigned to this role, or have knowledge of which ports members of this role has access to. This should be possible without having access to the source code, since the source code may be kept secret for different reasons, e.g. to avoid that other people copy the code.

4.2.4 Security Threats

A flawed authentication can be more dangerous than having no authentication, since it gives the users a false sense of security. They may more readily let the system handle sensitive information, when they believe that their data is protected from malicious persons, not knowing that a flaw in the authentication exposes the data to these persons. Therefore we will try to identify the different attacks our system may be exposed to, analyze the threats they may pose to the system and how this can be prevented.

Reconnaissance

To initiate an attack the intruder needs to know where to direct it, and which form of attack to use. He needs to know which services is running on the server and on which ports these are listening. This information can be obtained by doing a port scan, sniffing packets, or by social engineering.

A port scan can reveal which services a server provides and on which port numbers they are listening. This is done by sending a message to each port. From the response the attacker can determine which ports are used, and these ports can then be probed for weaknesses. There exists an abundance of tools that can be used to do a port scan on the net, e.g. *nmap* and *Netcat*. *nmap* will, given an IP address, report which ports are open, which services they support, and the owner of the daemon which runs the service. This is important since an intruder would gain the same privileges as the owner, if he is able to compromise the service. We use access control and 128 bit port numbers to prevent an attacker from gaining information about the server by doing a port scan.

As mentioned in section 4.2.2, 128 bit port numbers will greatly increase the time it takes to do a port scan. The time it takes to do a port scan depends on the bandwidth, the number of threads used, and the type of port scan. TCP connect() scanning is the fastest method supported by *nmap*.[3] If we assume that it is possible to scan one port in one millisecond it is possible to scan a server with 2^{64} ports in approximately a minute. With 2^{128} ports this takes:

$$\frac{2^{128}}{1000 * 3600 * 24 * 365} = 1.0790283070806 \times 10^{28} years$$

Beside this the attacker also needs to scan the UDP ports to do a full portscan. So even if an attacker has an extensive amount of resources he cannot gain any information from a portscan in reasonable time.

If an intruder is able to setup a computer on the network between the server and the client, he can monitor the traffic, and thereby discover which ports on the server the clients are communicating with. This can give him a general idea of which ports on the server provides services. This can either be done by using a packet sniffer, which is a tool that can retrieve all packets on the network, or by relaying the traffic sent to the server. This can be done by setting the MAC address of one interface card to the same as the servers. However, the intruder will then have to forward the packets he has intercepted to the server, if he wishes the attack to be unnoticed. Our system is not able to prevent a malicious person from discovering which ports the server is listening on if this form of attack is used. But since all traffic between servers and clients is encrypted it prevents the attacker from gaining any additional information from the attack. But even if an attacker gains knowledge of an open port on the server, he is unable to determine which service this port provides, since all packets which are not encrypted with the right key are dropped.

Social engineering is based on human interaction, so it is very difficult to set up a defence against it. An attacker will contact a user of the system usually by phone or email, and try to persuade the victim to hand over sensitive information (for example a password) or make the victim do something that will make the system vulnerable to an attack. Since our access control is based on

[3] See `http://www.insecure.org/nmap/nmap_doc.html` for details about this and other forms of port scanning

cryptography all an attacker needs to compromise a service is to obtain the key and the port number on which a service is running.

Denial Of Service (DoS)

A denial of service attack tries to deprive users of access to a service or resource, thus compromising availability. This is usually done by consuming bandwidth or CPU time, and thereby preventing other user requests from being serviced. DoS can be split up into three basic forms of attacks:

- Consumption of resources like bandwidth or CPU time
- Falsification of configuration information, f.x. changing routing information to redirect traffic
- Disruption of physical components, like a network cable

We will only look at the first kind of attack, since the latter two are attacks on network components and should be dealt with elsewhere.

A malicious person might try to exhaust resources on the server by sending an extensive amount of traffic to the server. Several methods exist that use the Internet Control Message Protocols (ICMP) to flood a host with traffic[4]. If a malicious person obtains knowledge of which port(s) a service is listening on, he can start sending malformed messages to these. Since we use CAC the service will decrypt all messages it receives and check if they are legitimate. Depending on whether or not the service is concurrent, the attacker can leave the particular service, or even the entire server, unable to respond. If the service spawns a new process to handle the incoming request, it is still able to process incoming requests while the first is being serviced. An attacker can then send multiple requests, consuming more and more resources on the server, which will be unable to handle incoming requests from other users, even if these are directed at other services. One way to circumvent this is to restrict each service to a certain amount of resources. If this is done, only the compromised service will be unavailable to the users. If the service only is able to handle one request at the time, the attacker can keep sending requests, so the service is unable to handle requests from legitimate users, but it will not affect other services provided by the server. This form of attack and attacks that use the TCP handshake (like syn flooding) requires that the attacker knows which port numbers are used, so 128 bit port numbers give some protection against this, but this protection is rather weak since an attacker, who can monitor the traffic, easily can obtain this information.

4.2.5 RPCgen

RPCgen is a tool to help the RPC programmer write RPC services by generating code, so the programmer only needs to focus on the implementation of the main features in the application. This means that the interface should be kept simple, and it should be well documented how to use the generated code. We need to update RPCgen to use our version of the RPC library, and at the same time make sure that the interface resembles the one used today. This is to ensure that

[4] For details of these attacks see [16]

existing RPC programs can be recompiled with our version of RPCgen with only minimal changes, and programmers who use RPCgen today can easily make the shift. Our version of RPCgen shall also be able to produce sample code for the server and client in a way, so that it is easy for the programmer to link this with the server and client stubs, in the same way as SUN RPC does today. This sample code should clearly document where the application specific code should be placed.

As mentioned, it should be easy for the administrator to generate new keys for the different roles, RPCgen should provide a method for doing this. To generate the keys, RPCgen needs information about which roles the service have, this can be provided by giving a file containing the program definition in RPC IDL, as it is described in the program definition file [5]. This will not disclose any secrets, since it only contains a definition of which versions exist for the service, which procedures and roles these consist of, and which procedures each role is allowed access to.

4.3 Requirement Specification

Below, we will summarize the requirements for our proposal for an extension of RPC which uses Cryptographic Access Control:

1. A new version of the RPC library must be designed. It should implement cryptographic access control on the individual procedure level in a manner that is as transparent to the users and programmers as possible, without changing the existing interface more than necessary

2. The access control must rely solely on possesion of keys, that is, it should be done without authentication of the client, since this scales badly

3. All access control must be performed inside the RPC library, so the code that handles the access control is the same for all RPC programs

4. As an additional security measure, the RPC portmapping mechanism must be replaced by an algorithm for calculating port numbers by use of hash functions and a shared secret

5. It should be possible for a running service to change the port number it is listening on with regular intervals

6. RPCgen should be extended to generate code that is compatible with the new RPC library

7. An extension to the RPC IDL syntax should be defined, making it easy to define the access control properties for each procedure in a program

8. As a consequence of the use of Cryptographic Access Control, all communication in the system will be encrypted, and therefore confidential. This must be implemented in a manner such that clients with access to the same procedure cannot eavesdrop on each other's calls

[5] See section 2.4.2

9. In addition to providing confidentiality, the system must provide integrity and availability where possible

10. RPCgen should contain routines to create the necessary keys, so a system administrator is able to change these easily

Chapter 5

Design

Essentially, the goal of the system is to restrict access to calling a particular procedure in an RPC program to the clients which are in possession of the correct cryptographic key. Besides the immediate effect of preventing unauthorized access to RPC programs, this will yield the benefits of confidentiality and integrity of the transmitted messages. It is important to keep the simplicity of the RPC protocol intact, and make the additions as transparent as possible to both RPC programmers, system administrators, and end-users.

In this section we will shortly describe how the server side is designed. We will explain what is done by the RPC library and what is done by the code written by the RPC programmer. We will then describe how the access control will be incorporated into the RPC library, so that is does not depend on the code written by the RPC programmer.

The RPC library contains methods to receive messages and reply to messages. The server stub written by the RPC programmer contains the dispatch functions, which are responsible for calling the right procedure after a request message is received, and the XDR filters which are used to serialize and deserialize the XDR data structures.

When the server receives a message, the RPC library creates a buffer to contain the XDR data structures and calls the dispatch function with the procedure number as argument. The dispatch function will then deserialize the arguments in the buffer, call the procedure associated with the given procedure number, and call the reply function in the RPC library, which will send the result back to the client.

Since a service will listen for incoming requests on a separate port for each role, we must verify that only members of a role are allowed to call procedures on the port associated with this role, and that they are not able to call other procedures than the ones accessible from the role. Each role will have a public/private key pair associated, which will be used in the cryptographic access control. Each client must encrypt a request with the public key and send the encrypted request to the server, which will verify that the request is encrypted with the right key by decrypting it with the private key. This means that only members of a role are allowed to call procedures on the port associated with that role, since each client that knows the public key is a member of the role. If the server is able to decrypt the message, it will call the dispatch function. To ensure that a client is only able to call procedure accessible for members of its role, it will be natural to limit requests on each port to calls to the procedures which the associated role has access to. This means that there must be one dispatch function for each role instead of one for each version, as it is in the original implementation of RPC. Unfortunately, this creates some restrictions to how the RPC programmer writes his dispatch function, that we cannot verify. E.g. a RPC programmer may write a dispatch function for each version, as he would do with the original implementation of RPC. This is illustrated in figure

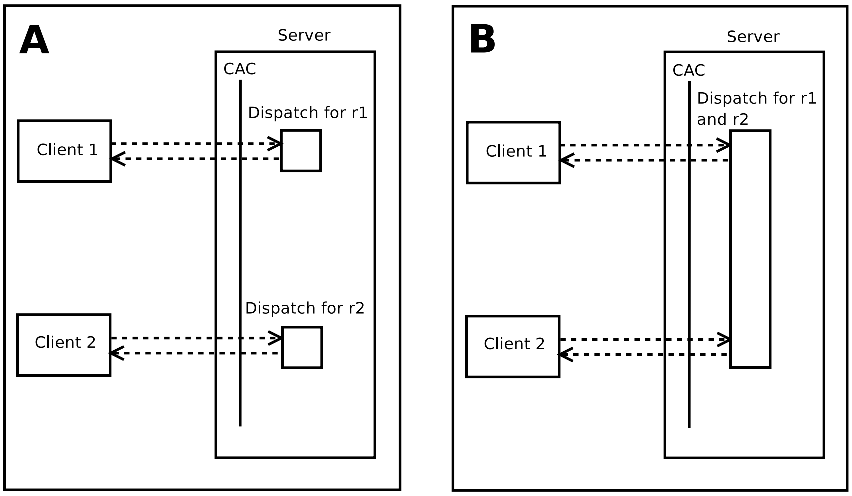

Figure 5.1: A shows a RPC program with a dispatch function for each role, and B shows a RPC program with only one dispatch function

5.1, where we have a RPC program with two roles, r1 and r2, where client 1 is member of r1 and client 2 is member of r2, but not vice versa. In A the program has a dispatch function for each role, and in B the program only has one dispatch function. As shown we will in both cases verify that client 1 is member of role r1 and client 2 is member of r2. In B we have no way of knowing which procedure is called after the dispatch function is called. We will therefore use ACLs to verify, that a client has access to the procedure it tries to call, before the dispatch function is called. Each role must have its own ACL, consisting of the list of procedures which members of this role has access to. Since we already know which role the client is a member of, and the fact that the server was able to decrypt the request proves that the client actually is a member of this role, we only need to verify that this role has access to the procedure the client tries to call. Since the access rights of roles are constant in a given RPC program, our reference monitor will have a consistent overview of the access control state, which means that the addition of ACLs does not change the scalability of our system in a distributed system. Therefore, we find that it is preferably to introduce this additional layer of security to protect the RPC programmer, even though it does conflict with the pure cryptographic access control model.

5.1 Protocol

In this section we will give a detailed description of the protocol we have developed. We will explain how the system will restrict access and how confidentiality and integrity are ensured.

In our system access rights are assigned to roles. Each role has a set of procedures which its members are allowed to call. To become member of a role a user has to obtain the public key for this role and the shared secret which is used to generate the port number. These can be obtained from another member or from the system administrator. Because membership is determined by knowledge of these two things, there exists no way to find all the members of a particular role. Of course the system administrator will have a list of the users he distributed the keys and secrets to, but he cannot know if these have distributed them further. E.g. in a large company the system administrator may choose to give access to one person in each department, this person can then distribute access to the users in his department who need access.

The members of a role know the public key, while only the server knows the private key. This means that only the server can decrypt messages encrypted with the public key, but all the members of a role are able to decrypt messages encrypted with the private key. As described in section 4.2.3 we have introduced mixed mode cryptography to ensure the confidentiality of the reply from the server, which will be encrypted with a session key. This will also improve performance since it is less expensive to encrypt and decrypt with a symmetric cipher than with an asymmetric one. The client will create a new session key for each call to the server and send this to the server along with the request. It will of course decrease the performance of the client to create a new key for each call, but the server will only benefit of the use of a symmetric cipher since the most of the encryption and decryption of the server is done with the symmetric cipher. When the client wishes to call a remote procedure it will first create a session key, use this to encrypt the request, encrypt the session key and a nonce with the servers public key, and send these to the server. The server will use its private key to decrypt the session key and nonce, remember the nonce to guard against replay attacks, decrypt the request, service this, and send a reply back to the client.

The part of the message which contains the actual request and is encrypted with the session key, needs to hold the data needed by the server to identify the procedure, the client wishes to call, together with the arguments to this procedure marshalled in XDR. To ensure the integrity of this data we hash it, and encrypt both the data and the hash value of the data, before the request is sent to the server. The server will upon reception verify that the hash value is correct. The same will be done with the reply to the client. Besides the two encrypted parts the message from the client to the sever will also contain an id in clear text. The server will include this id in the reply to the client, which will identify the reply from this id. The structure of the messages is shown in figure 5.1.

Since the access control must be as transparent as possible to the user, we will do all the client side encryption and decryption in the functions the client stub uses to interface with the RPC library. The only thing which must be done by the client is to calculate the current port number the service is listening on, and pass this and the public key to the client stub. On the server

Request : $X_{ID}|\{K_S|N\}_{K_{public}}|\{N_{proc}|Args|H(N_{proc}|Args)\}_{K_S}$

Reply : $X_{ID}|\{\text{Res}|H(\text{Res})\}_{K_S}$

- X_{ID} is the session ID used by RPC to match calls and replies sent over a UDP transport
- K_S is a random symmetric session key generated by the client
- N is a nonce
- K_{public} is the roles public key
- N_{proc} is the procedure number
- Args is the arguments for the remote procedure
- H is a hash function

Figure 5.2: The request and reply messages

side it should still be possible to use dispatch functions written for the original RPC implementation, therefore we will place the encryption and decryption routines in the functions in the RPC library which are responsible for sending and receiving messages. By placing all cryptography and access control inside the RPC library we also ensure that all services that use our version of RPC will use access control, and that the access control used is the same.

Figure 5.3 shows how we have incorporated the encryption and access control into the RPC protocol. As shown in the figure the main differences between our version and the original version are: The client will now generate the port numbers instead of asking the portmapper, all messages sent between the client and server stubs are encrypted, and the server stub will verify that the client has access to the procedure before it is executed. In the figure it is not shown that the server will verify the message being wellformed after it is decrypted. When the server has verified that the client possesses the correct key to call the procedure it will check that the role, which has access to the port it received the request on, has access to the procedure which is being requested.

One could easily be led to believe that our access control is an implementation of the RBAC model, but this is not the case since the RBAC model clearly states that it should be possible to determine all the members of one role, and which roles any given user is member of. This is not possible in our model since role membership is acquired by possessing the right key and the secret used to generate the port number, and any member of a role can distribute these to other users. This means that there is no simple centralized method to obtain all the members of one role, or which roles a given user is a member of.

Our access control implements the DAC model, in the way that each role has a list of procedures to which the members have access. Each member of a role may pass membership, and thereby the access rights, of a role on to other

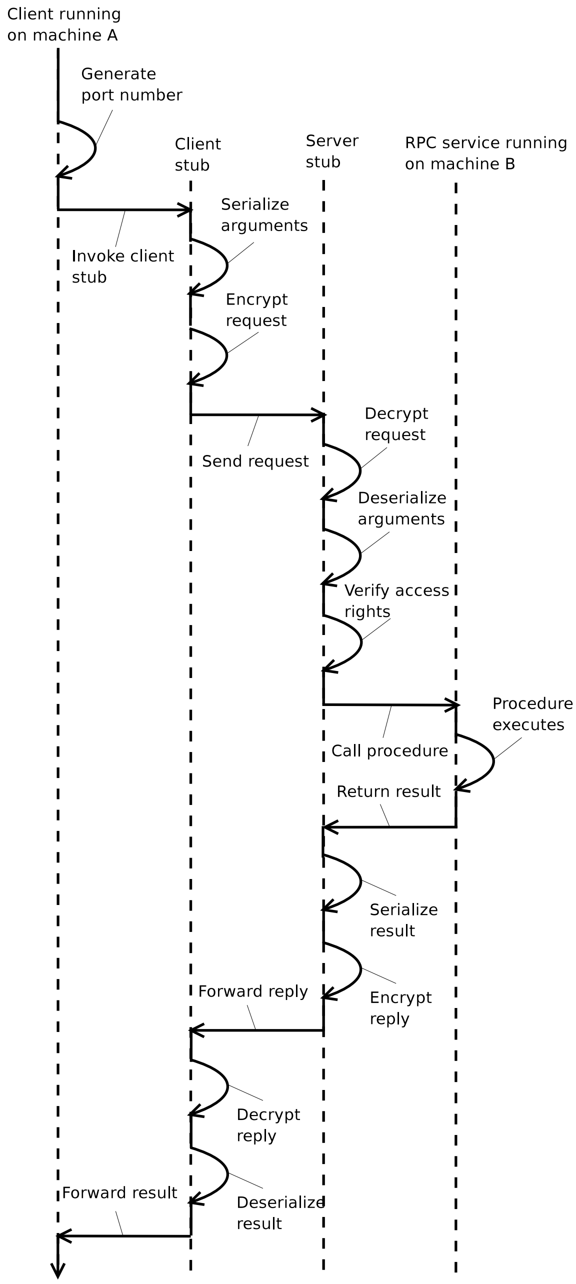

Figure 5.3: The remote procedure call model with cryptography

users.

5.1.1 Port Number Generation

To access a remote procedure the client need to know on which port the specific RPC service is currently communicating. Since we do not want to use the portmapper, we need some way to let the client calculate the port number. There are three requirements to this. The port number must be able to change at a regular interval without causing breaches in ongoing communication, and people who are not a member of a particular role, should not be able to calculate the port number the service uses to communicate with this role. Finally we need some way to handle hash collisions.

To ensure that no ongoing session is destroyed when the service changes port number, there must be an overlap where the service is listening on two ports for each role that has changed port number. If the old port has not received any incoming requests after a predefined amount of time, it will be closed.

We wish to give the RPC programmer the opportunity to change the way in which the port number is generated, therefore we have placed the generation of the descriptive string outside the RPC library. If the RPC programmer wishes to use our method he can use RPCgen to generate sample code, that will generate the string, or he can create his own code to generate a string. The string must be passed to the function that creates the client or the server, which then will hash the string and concatenate this with the hash of the public key and hash this to get the port number. This provides high flexibility to the RPC programmer, since he is able to decide how often a service needs to change port number. It is even possible to let the roles of one service change port numbers at different intervals, e.g. it may be desirable to have a role, which always runs on the same port number, i.e. a role for guests, while the more privileged roles need to change port number once a day.

If a service creates a port number which is already in use, a new port number is created by using the port number which led to a collision instead of the descriptive string. So first we create a port number by concatenating the hash value of the descriptive string with the hash value of the key. This is then hashed and we make a check to see if the port number is already taken. If this is the case, we repeat the algorithm but this time it is the hash value of the colliding port number which is concatenated with the hash value of the key. This is illustrated in figure 5.4. The algorithm will be repeated until an unused port number is generated or until the algorithm has run a predefined number of times. Since we use 128 bit port numbers it is highly unlikely that a collision will occur, and extremely unlikely that it will happen more than once. If a hash collision occurs on the server it will mean that some clients will try to call a service on the wrong port. This would cause the server to drop the request since it is encrypted with the wrong key. The client will then wait for a reply until its timeout is reached, whereafter it will generate a new port number in the same way as the server does in case of a collision, and send the request to this port number. The client has a predefined number of times it will try this before it returns an error.

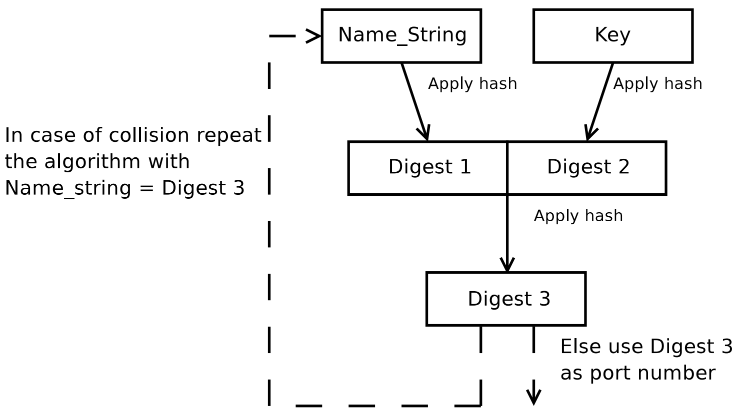

Figure 5.4: Port number generation

5.2 The RPC interface

In section 2.4.3 we described the API to the RPC library. This section will describe how it will be changed in our implementation. Since we are only designing a prototype, only the functions used by RPCgen (the ones referred to as the lowest layer in section2.4.3) is changed. We have tried to keep the interface from the original RPC library, so the code the RPC programmer has to create will resemble the code used in the original RPC. Our prototype will only support UDP so we will concentrate on the interface functions associated with UDP.

On the server side we have added one function and removed the function which was used to remove previous mappings of the service from the portmapper tables, `pmap_unset`.

svcudp_create Since our design will assign port numbers to each role, dynamically, it will no longer be possible to specify a socket, instead the function will always create a socket, calculate the port number on which the service is available for the given role, and bind the socket to this port number. To calculate the port number the functions need to know the descriptive string belonging to the users current role and the public key, this means that the RPC programmer must generate the descriptive string and pass it to `svcudp_create` as a parameter. Besides the port number `svcudp_create` needs a role number and the private key for this role to create the transport handle, these are also given as parameters. Because each role must connect on its own port number, it will be necessary to create a transport handle for each role, so `svcudp_create` must be called for each role.

pmap_unset This function is no longer needed since we do not use the portmapper.

get_role_name This function can be used to read a role name from a file. It is only meant as a help to the RPC programmer if he wishes to use the role name to create the descriptive string used to generate the port number. It will take the role number and file name as parameters and return the name associated with the given number.

svc_ register The primary purpose of this function was to register the service with the portmapper, but this is no longer needed. Instead this function will be used to specify which procedures each role has access to, so it must be called for each role, and the RPC programmer must specify which procedures the role has access to and pass these as a parameter. Since we register each role instead of each program version, we need to pass the role number instead of the program and version number.

The most significant changes to the original interface functions used by the server, lie in the change that we need to create a transport handle for each role, instead of only creating one for each service, and that the RPC programmer needs to generate the descriptive strings and the keys, and specify the procedures each role has access to.

The only change in the client side interface is introduced because the port number must be generated instead of obtaining it by querying the portmapper.

clntudp_ create It will no longer be possible to specify the port number in the socket data structure, since `clntudp_create` will always generate the port number from the public key and descriptive string and set the port number. It is no longer necessary to pass the program and version number as parameters, because we will not use these to query the portmapper, instead the descriptive string and the public key is sent along. The same changes will be made to `clnttcp_create`.

clntudp_ call The parameters for this function will remain the same as in the original rpc. `clntudp_call` will still be responsible for sending the request and receiving the reply. Besides this it will also take care of the encryption and decryption. Also it will be changed so the request messages will be structured as described in section 5.1.

clnt_ destroy No alterations will be made to this function.

The client side interface is almost the same as in the original. The only difference is that the RPC programmer needs to generate the descriptive string and pass this to `clntudp_create` instead of the program and version number. Besides creating the arguments for the functions mentioned above and calling them, the RPC programmer must create the dispatch function, which is responsible for decoding the arguments for the procedure, calling the right procedure, and calling the function which will send a reply to the client. The dispatch function uses the procedure number to call the right procedure. This means that each dispatch function must not control more than one version, because this could lead to one dispatch function which controls two procedures with the same number.

The RPC programmer can also create a series of services, which have the same roles and read the keys from the same files, thereby making the key distribution easier. However. he must be aware that two roles with the same key must not have the same descriptive string, since this will lead to hash collision. The system administrator is also able to associate different roles with the same key, by storing the same key in the files used by the roles he wishes to associate.

5.3 RPCgen

Here we will describe the features we wish to add to RPCgen and the changes that are necessary in order to make RPCgen able to parse the new keywords, we will add to RPC IDL, and generate code which will use the new syntactic constructs in the RPC library[1]. We need to change the existing calls into the RPC library to call our extended versions of the interface functions instead, and add some new features to RPCgen. Original implementations of RPCgen can create a server and client stub, filters to serialize data, and a header file. Besides this more recent implementations can generate sample code, which shows how to interface with the client and server stubs. These can be used as a starting point for the RPC programmer.

Our version of RPCgen should be able to generate keys to be used in our access control. We will use a cryptographic library to generate a public/private key pair, which is stored in two files; one for each key. This needs to be done for each role defined in the RPC IDL program definition file. The public keys will then be distributed to the users, so that each user receives the keys associated with the roles of which he is a member. The private keys are given only to the server. To give the system administrator the possibility to replace the keys used by a role, we will add a flag to RPCgen, which will cause only the keyfiles to be created, these can then be distributed to the clients who will replace the original files.

As described in section 4.2.2 RPCgen will generate port numbers based on the method used in Freenet to generate keys. To ensure that it is not possible to calculate the port number from the public key alone, the descriptive string must be calculated on a shared secret between the server and the members of a role. Besides this the system administrator need some way to change this secret if it is compromised, and to ensure that old members of a role does not know the secret. We have chosen to use the role name as a shared secret, and this should be read from a file or given as a parameter, so it is possible to change it. The string will also include the program and version name and number. These together with the role name and number ensures that the string is unique for each role. To change port number at a regular basis we also include the current date in the string, so it will change every 24th hour. Thus ending up with a string on the form:

<*hostname*>/Program_<*Program number*>_Version_<*Version number*>/
<*Program Name*>/<*Version name*>/<*Role name*>/<*Role number*>/<*Date*>

To allow the system administrator to change the shared secret used to create the strings, our version of RPCgen will create a file, which will map role numbers into role names. This means that whenever a new port number is generated the role name associated with the role, which is about to be assigned a new port number, is found in this file. Since this file will be written in clear text, the administrator can either choose to change the file in an editor or use RPCgen to create a new file, but then he needs to change the RPC IDL program definition file instead. Whenever a role changes name the administrator must distribute this to the members of the particular role, but as with key distribution the system to distribute this is beyond the scope of this project.

The changes to the RPC library means that both the generated server and client code needs to be changed. For the server we will need to create multiple

[1] These are described in section 5.2

transport handles for each version instead of using the same for all procedures. `svcudp_create` must be called for each role in each version. Before each call to `svcudp_create` we must generate the descriptive string, according to the algorithm described in section 5.1.1, and the name of the files in which the roles public key is stored. For the call to `svc_register` we must specify the procedures the role has access to.

In order to enable the RPC programmer to specify role access assignment, we have chosen to extend the RPC IDL with a keyword specifying the relationship between a key and the set of procedures to which it grants access. For each version of each program defined, it is possible to define a list of *roles*, i.e. each role defines a set of procedures in the program, which are accessible using the key associated with the role. To illustrate this, we give an example of a simple RPC IDL definition using the extended syntax:

```
program PRINT_PRG {
  version PRINT_VER {
    int print_job (...) = 1;
    int add_printer (...) = 2;
    int delete_job (...) = 3;
    role USER_ROLE {1} = 1;
    role OPERATOR_ROLE {1,3} = 2;
    role ADMIN_ROLE {1,2,3} = 3;
  } = 1;
} = 0x20000001;
```

In each role, the list in brackets indicates the set of procedures to which the role is granted access. Each role is declared with a role name and number, which are used internally in the same manner as program and version numbers in the original RPC protocol. The full syntax of the extended IDL is given in appendix A.

5.4 Summary

We have described how we will incorporate access control into RPC. Each service will be divided into roles, where each role will have access to a subset of the procedures provided by the service. Each role will have a port associated, where requests from members of this role are accepted. To ensure that only requests from the members are accepted, the roles each have a public/private key pair, and all requests must be encrypted with the public key. The server can then verify role membership by decrypting the request with the private key. Role membership is obtained by possesion of the public key associated with the role, and knowledge of the shared secret between the server and the role. The shared secret is used in our naming scheme, where we generate port numbers from a descriptive string and the roles public key. This is done to enable clients to identify on which port they are able to communicate with a service.

Integrity and confidentiality is ensured by applying hash values to each message and encrypting them. As mentioned the cryptographic access control is done with an asymmetric cipher and the messages are encrypted with a symmetric key.

We have also described how we will change the RPC API, and how RPCgen will be changed to make use of the changes and new features added to RPC.

Chapter 6

Implementation

We will now describe the implementation of our prototype of Remote Procedure Call with Cryptographic Access Control (CACRPC). The implementation is based on Sun Microsystems' version 4.0, the source of which can be obtained from `ftp://playground.sun.com/pub/rpc/`. This rather outdated version of RPC was preferable to the one that ships with a modern Linux distribution, since the latter is integrated with the GNU C library. This makes it difficult to introduce changes without affecting some other part of this very complex system.

The Sun implementation, which is from 1989, first had to be upgraded to comply with modern C standards, and any references to old system calls had to be removed. This process has not been entirely completed, so some warnings due to uses of deprecated functions may occur at compile time. The age of the code also means that some features are lacking, especially the Secure RPC authentication flavor implementation has been left out due to the export restrictions on the DES algorithm enforced by the U.S. government at the time. This is not a problem, however, as this part of the RPC standard is not important for our purposes. We have also backported some improvements made to the implementation of RPCgen in GLIBC to the old Sun version. These include the sample code and makefile generator components.

6.1 Choice of Cryptographic Algorithms

There exist a great selection of different cryptographic libraries that implement the most popular algorithms and finding the one best suited for a particular application is not a trivial task. We have chosen to base our prototype on the Nettle library by Niels Möller [17]. It has the advantages of being very low-level, in that it does not do automatical algorithm selection, memory management, and so on. This means we can apply encryption directly on the data in the XDR data buffers used by the existing RPC implementation. Nettle is also relatively simple to use, and has support for the needed functionality.

We have chosen the AES algorithm for symmetric encryption, since it is a well-proven algorithm with support for the key sizes required in modern systems. All encryption is carried out in CBC mode, in order to prevent block manipulation. Asymmetric encryption is done using the RSA algorithm. This is in fact the only asymmetric algorithm supported by Nettle at present, but this is not really a problem, as it is an old algorithm that has withstood cryptanalysis for almost thirty years. One downside is that it requires fairly large keys in order to be considered secure. Since no security system is stronger than its weakest part, the most efficient use of resources is obtained if all the used components are approximately equally difficult to attack through brute force. RSA security claims that 2048 bit RSA keys are equivalent to 112 bit symmetric keys, and considers data encrypted with such a key to be secure until the year 2030. Since the minimum key size of AES is 128 bits, we cannot obtain perfect equivalence. This could be achieved by increasing the RSA key size to 3072 bits, if desired.

The hash function SHA-256 is used both for port number generation and for ensuring integrity. As the name suggests, it generates 256 bit hash values (since the port numbers used are 128 bits long, the hash value is truncated when used for this purpose). Due to the attack known as the birthday attack, hash values must generally be twice as long as what one immediately expects. This is due to the fact that an adversary only need to find two random messages that hash to the same value in order to potentially be able to break a system reliant on a hash function. This is much easier than finding another message that hashes to the same value as a given message. For this reason, the hash value size of 256 bits is comparable in strength to the 128 bit keys used in AES. Being extra cautious when choosing a hash function is also very important, since a recent paper by Chinese cryptographers Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, reveals a weakness in one of the most popular hash functions, SHA-1, making it 2000 times faster than brute force to find a collision. SHA-256 might be vulnerable to similar attacks, but with its much larger hash value length, it should be secure for the foreseeable future even if such problems do exist. An attacker finding a collision is primarily a problem if he can use it to compromise message integrity. Finding colliding port numbers is not really a security risk.

6.2 Overall Program Structure

The central parts of the source code are organized into two directories; one for the RPC library and one for RPCgen. In addition to these, there are also some demonstration programs, source code for various standard RPC services, an incomplete version of secure RPC (due to the above mentioned export restrictions). These will not be described in this section and are left unchanged from the Sun distribution of the source.

There is no real distinction between the client side and server side parts of the RPC library. When writing an RPC program, the programmer must include the header file `rpc.h`, and link against the static library `librpclib.a`. However, the source code is organized into separate files for client and server specific code, so some separation is possible. We will therefore describe these two parts of the RPC library separately in the following two sections. The implementation of the most important library functions will be illustrated using UML sequence diagrams. It should be noted that the diagrams do not adhere strictly to the code; some details have been omitted for clarity, and the diagrams are kept in the object-oriented style for which UML is most suited, even when the C code deviates from it. To save space, system calls are represented by messages to self in the diagrams, rather than introducing communication with a “system” object.

We have added some common wrapper functions for interfacing with the Nettle library. These are used both by the client and the server, and are found in the files `common_cac.h` and `common_cac.c`. They will not be described in detail here, as they are fairly simple. In the diagrams below, they are represented by the static class Crypto.

6.3 Client Side

The main data structure of the client is declared in `clnt.h`. This is the `CLIENT` structure, which is initialized by calling the `clnt_create` function described below. It mainly consists of an array of function pointers, which are initialized to point to various operations that can be performed on the client. The most important is `clnt_call` which is used to call an RPC procedure. Others include a destruction routine, an error printing routine, etc. The reason that function pointers are used, is that the functions are transport protocol specific. Our implementation only supports the UDP protocol, therefore the diagrams below actually describe the UDP versions of these functions. The UDP-specific functions are found in `clnt_udp.c`, whereas the generic `clnt_create` wrapper function is found in `clnt_generic.c`.

6.3.1 clnt_create

This function, which is illustrated in figure 6.1, creates a client structure. It first looks up the IP address of the provided server hostname using the system call `gethostbyname`. The Nettle random number generator is then initialized. We then read an RSA public key from the file specified by the caller. This key is used to encrypt all message headers when calls are later made using this client structure. We now generate a port number by hashing the argument `name_string` concatenated with the public key. The present implementation does not support collision detection on the client side. Finally, a UDP socket is created and stored in the client structure. The client structure is returned.

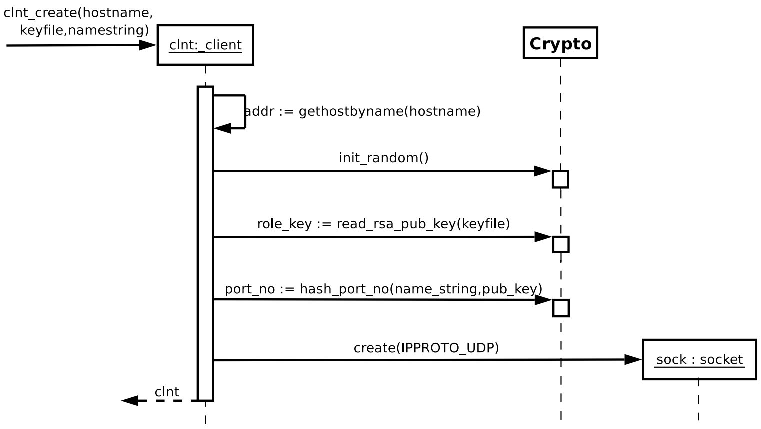

Figure 6.1: The `clnt_create` function

6.3.2 clnt_call

This function is described in figure 6.2. It is called by a client stub when the user wishes to call an RPC procedure. It is responsible for marshalling all arguments and settings for the procedure, sending these across the appropriate network socket, waiting for the proper reply, and finally unmarshalling the reply. We introduce the additional steps of adding encryption to the call message, and decrypting the server's reply. The arguments of the function are the following: The number of the procedure to be called, the arguments in their unencoded form, and pointers to two XDR filter functions that are able to encode the arguments, and result of the function, respectively.

The function first sets up some cryptographic parameters by generating a random symmetric session key and an initialization vector. An XDR buffer object for encoding data is also created, and the xid is placed into this buffer. After which a timestamp is generated, and the asymmetric role key is used to encrypt it and the session key, placing the resulting ciphertext into the buffer. The initialization vector, procedure number, and arguments are now encoded into the buffer. We have chosen to use timestamps instead of nonces as indicated in the design specification, since this eliminates the need to maintain a table of used nonce values. Then, we pad with zeroes until the buffer length is an integral multiple of the symmetric algorithm's block size, and a digest of the buffer is added to the end of the buffer, at last the session key is used to symmetrically encrypt the part of the buffer that is not already encrypted (using the key and initialization vector chosen at the beginning). The entire buffer contents are then sent to the server. A call is placed to the `select` system call, waiting for a reply from the server.

Once a reply arrives on the socket, a check is made to verify that it starts with the same xid as the original message. An XDR buffer is created and the reply is copied to this. The initialization vector, used by the server to encrypt the reply, is read from the buffer. The rest of the buffer is decrypted with this initialization vector and the session key. The `xdr_replymsg` filter for decoding the reply is invoked, taking into account the possibility that the reply is an error message. `clnt_call` is also responsible for calling the result-decoding XDR filter supplied by the client stub. Finally, we calculate a digest of the reply buffer, comparing it with the one that is stored in the buffer itself. If they do not match, we return an error, otherwise we return the result.

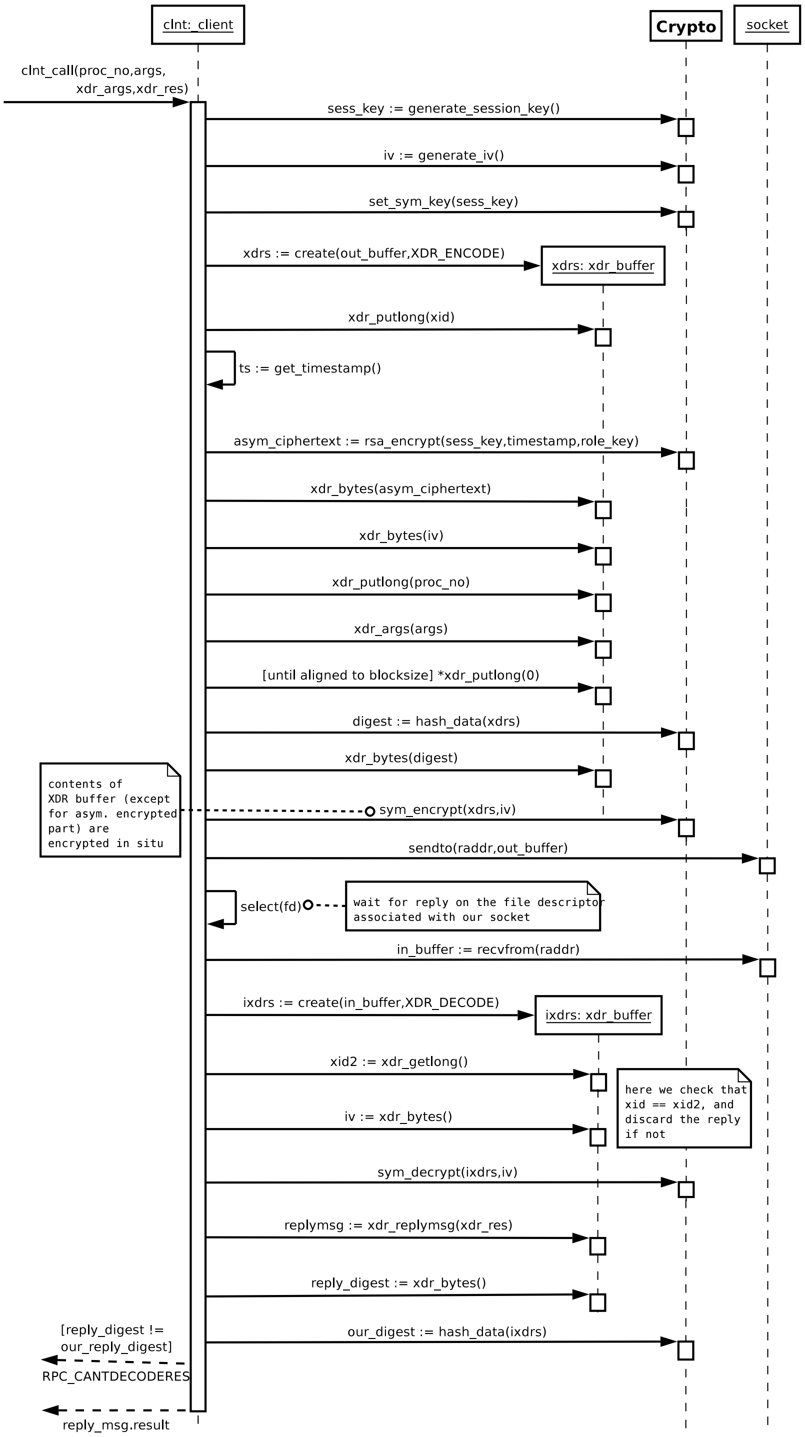

Figure 6.2: The `clnt_call` function

6.4 Server Side

The server uses a transport handle to contain data needed in the communication with the clients and to pass this data between functions. This includes which socket the service is receiving on, the port number to which this socket is bound, the address of the client which is currently communicating with the service, and a struct of function pointers. The server uses these functions to do the actual communication, like receiving messages, replying to these, extracting data from the messages and to handle memory deallocation. These functions are defined in the RPC library and are transport protocol dependent. The server stub is responsible for creating this transport handle, which is done by calling a function in the RPC library, e.g. `svcudp_create` if the UDP protocol is used. Besides the transport handle the original server implementation uses a list for keeping track of which dispatch function is used by each program version. The server stub should call `svc_register` for each version of each program, which builds this list and registers the program with the portmapper. The dispatch function is a callback function, which should extract the arguments to the procedure call from the request, call the procedure requested by the client with these arguments, and return the result to the client. The function pointers contained in the transport handle should be used to extract the parameters and return the result. But since the dispatch function is declared by the RPC programmer, he may choose to do it in another way. When the transport handles are created and the different program versions are registered, a call to `svc_run` will start the service. `svc_run` is an infinite loop which makes a `select` system call, which returns when one or more sockets change status. After which a `svc_getreqset` is called, which first receives the message, by calling the receive function pointed to by the transport handle. It will then find the dispatch function associated with the program version specified in the received message, and call this. If the program version is not found in the list, it will return an error with the highest and lowest version of the program the server supports.

Since we want each role to connect on its own port, our implementation will need one transport handle for each role instead of one for each version. This means that we can use the transport handle to contain information needed for the access control. We will also move the pointer to the dispatch function inside the transport handle instead of using a list. This is done in order to prevent clients from calling a different dispatch function, which in the old design could be done, by changing the program and version number in a request.

The extension of port numbers from 16 bits to 128 bits is only simulated in the current implementation - the actual port numbers passed to the system calls handling the sockets are the first 16 bits of the port number. However, the full 128-bit port number is stored internally in the RPC library data structures, in order to verify that the connecting client has actually generated the port number using our hashing algorithm. A full implementation of longer port numbers will of course require a redesign of the transport protocols, which is beyond the scope of the project.

The transport structure, which is named `SVCXPRT`, is declared in `svc.h`. The UDP-specific functions, which we have reimplemented, are found in `svc_udp.c`.

6.4.1 svc_create

This function is illustrated in figure 6.3. The purpose of the function is to create a transport structure with the appropriate parameters. First a UDP socket is created, then the transport object itself. An RSA key pair is read from the file specified in the argument `keyfile`. The public key is used to create a port number as described in section 5.1.1. The socket is now bound to the generated port number using the `bind` system call. If this is unsuccessful, due to the port being in use already, the hashing process is repeated until an available port number has been found, or the maximum number of iterations have been run. This number is defined by the macro `MAX_COLLS` and is set to 3 in the current implementation.

Once the port number has been chosen, and the socket successfully bound, we initialize the Nettle random number generator and save the private key of the keypair in the transport structure (this is the key that will be used to decrypt the headers of all incoming calls on the port of the transport.) Finally, we `register` the transport, i.e. create an entry in the server's global table, thus associating the file descriptor number with the transport handle. The function returns the created transport structure.

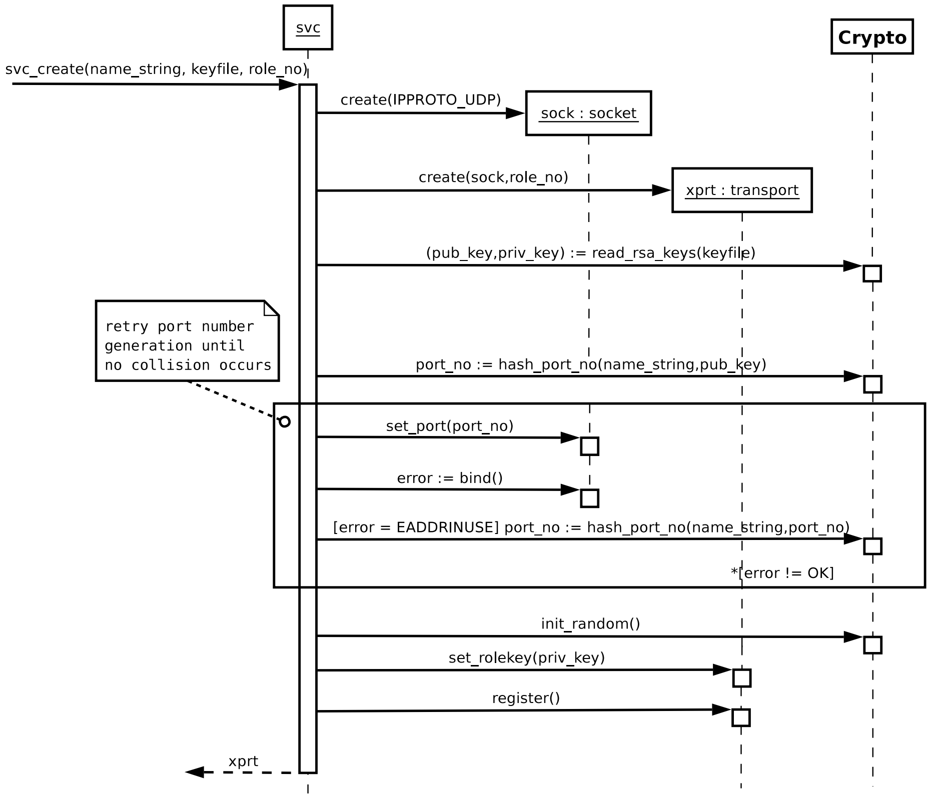

Figure 6.3: The `svc_create` function

6.4.2 svc_register

This function associates a dispatch callback function and a list of procedure numbers with an existing transport. The dispatch function is provided by the RPC programmer and handles the execution of the body of the appropriate procedure code when an RPC call is invoked. The procedure number list contains all the procedures which are to be accessible on this transport, and is used to build the ACL-like construct mentioned in section 5.

6.4.3 svc_run

This function is illustrated in figure 6.4. The `svc_run` function is the infinite loop that the server enters once all transports and programs have been properly registered and configured. It will listen for incoming data on all relevant file descriptors using the `select` system call. Whenever data is received, this call will return the set of file descriptors that have new data. The `svc_getreqset`

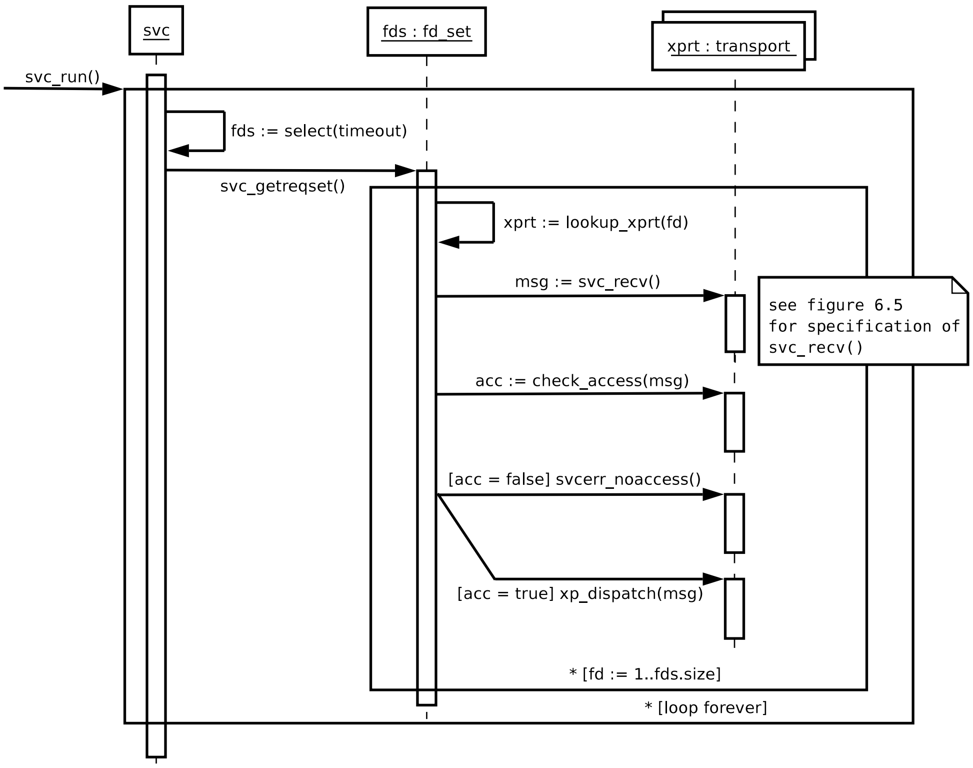

Figure 6.4: The `svc_run` function

auxiliary function will now iterate over each of these descriptors, first looking up the corresponding transport structure in a table, then calls the `svc_recv` function on the transport handle. This function, which is described below, returns the RPC call message received. The contents of the message are examined, verifying that the procedure number called is in the list of allowed procedures for the role of the current transport. If this is not the case, an error reply message is generated and sent by a call to the function `svcerr_noaccess`. Otherwise, the dispatch function registered for the active transport by the RPC programmer is invoked with the message itself as the parameter. The dispatch function is responsible for generating the reply at this point. Once the file descriptor loop terminates, the main loop runs again, and we call `select` again, waiting for the next call to arrive. In the prototype implementation the server is not able to change port numbers dynamically. It generates a port number on start up and will not change this, before it is restarted. This is unfortunate since it means that the server must be restarted every time the method to generate the port numbers dictates, that the port numbers is changed. In a full implementation, this should of course be implemented.

6.4.4 svc_recv

This function is illustrated in figure 6.5. As mentioned above, this function is called by `svc_run` whenever incoming data is detected, in order to obtain the RPC message sent. First, the `recvfrom` system call is called on the socket

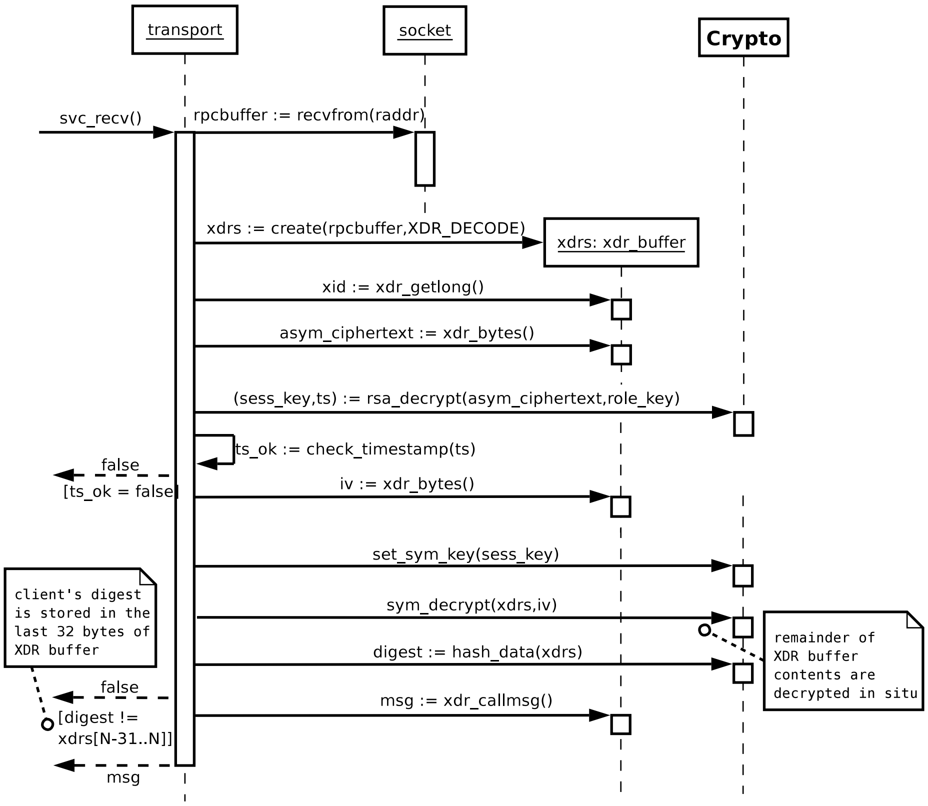

Figure 6.5: The `svc_recv` function

stored in the transport structure. The resulting raw data buffer is wrapped in an XDR structure, enabling the data to be read from it in a structured way. The first thing read is the xid, which is stored for later. Then, an array of bytes from the XDR buffer is read into another buffer. This is the asymmetrically encrypted session key and timestamp. These are decrypted using the role key. It is now verified that the timestamp is recent, returning if it is not. The initialization vector is now read from the XDR buffer and the rest of the buffer is decrypted using the vector and the session key. The symmetrically encrypted part of the buffer contains a hash value at the end, which is 32 bytes long. To check the integrity of the message the hash value of the whole buffer except for these 32 bytes is created, and the two digests are compared. If they do not match, `svc_recv returns`. If they do, the function `xdr_callmsg` is called. This function copies the contents of the XDR buffer into a message structure (here we read the actual call information, i.e. the arguments and procedure number). The resulting message structure is returned on successful completion.

6.4.5 svc_reply

This function is illustrated in figure 6.6. It is normally called by the dispatch routine callback function (written by the RPC programmer). The function is responsible for encoding, encrypting and sending a reply to an RPC call. The input parameter is a message structure, which contains the result of the call, as generated by the dispatch function. First a random initialization vector is generated, then an XDR buffer, which is ready for encoding data. The first data to be encoded is the xid, which was stored in the message structure by the `svc_recv` function previously. This enables the client to match the reply to the correct call. After this, the initialization vector is placed into the buffer, followed by the contents of the call message. Now, a hash of the buffer contents is calculated (not including the xid and initialization vector) and put into the buffer. Then the data as well as the hash value are encrypted directly in the buffer, using the same session key sent to the server by the client at the time of the call. At last the `sendto` system call is invoked in order to transmit the contents of the buffer back to the client.

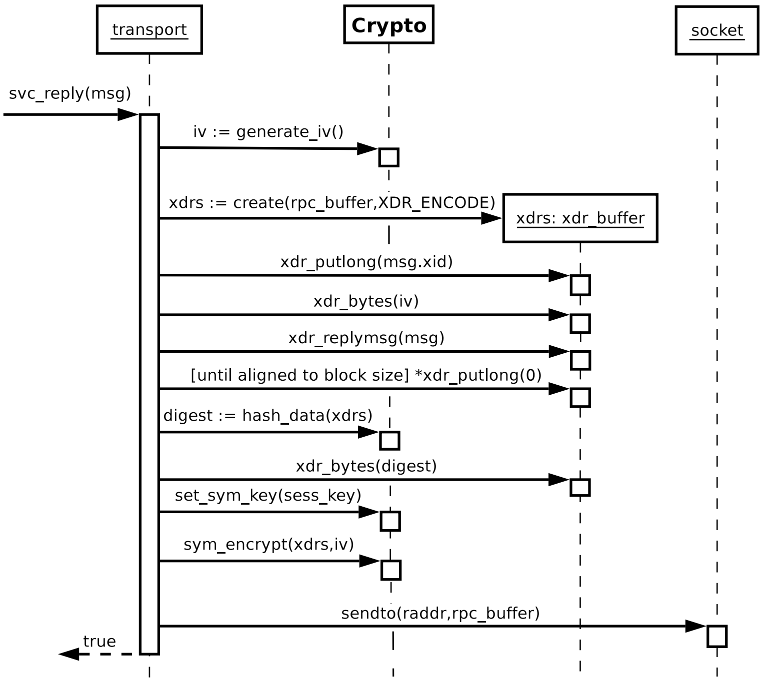

Figure 6.6: The `svc_reply` function

6.5 RPCgen

This section will explain how our version of RPCgen is implemented. We will first describe the basic organization of the code. RPCgen is divided into files according to the different output files it can generate. Each file contains methods to create one of the output files. We will describe the function of each file, the files we have not changed or only made minor changes to will be described on a superficial level, like it will not be described how include statements and other simple code is written to the output files. The output files we have created and the changes we have introduced to the existing output files will be described on a more technical level.

rpc_main.c contains the main function, the `parseargs` function, some functions to open the input and output files, and the functions which creates the output files, by calling the functions defined in the different other files. The main method of RPCgen first reads the parameters given to the RPCgen call. This is done with the function `parseargs`, which checks that all the parameters are legal and that they are used in a legitimate way, Otherwise, it returns zero and the main method prints the usage message. `parseargs` is also responsible for setting the appropriate flags according to the parameters and passing the input and output files if these are specified. The correct use of the flags and which operations are performed when the different flags are used is explained in B. According to the flags passed to RPCgen the main method will call the functions which will create the output files.

rpc_parse.h contains the data structures used to build the parse tree from the RPC IDL definition file. For each data type or program declaration a `definition` data structure is built. There exist 6 kinds of definitions: Struct, union, type definition, enum, program, and constant. The `definition` data structure contains a name, a definition type, and a struct, which contains the relevant data for the definition.

rpc_scan.c contains functions to scan the RPC IDL definition file.

rpc_parse.c contains functions to build the parse tree.

rpc_util.c contains various auxiliary functions.

rpc_hout.c contains functions to write the header file, see appendix C.3.

rpc_cout.c contains functions to write the XDR filter routines, see appendix C.2.

rpc_clntout.c contains functions to write the client stub, see appendix C.4, it will for each procedure in each version print a function that calls the remote procedure with `clnt_call`. Each function will have the same name as the remote procedure it calls appended with the program version number.

rpc_svcout.c contains functions to write the server stub, see appendix C.5. If RPCgen is called with *-m* as parameter only the dispatch functions is generated, else there will be generated a main method for the serverstub, which registers the roles and starts the service, with a call to `svc_run`.

First the main method of the server stub is generated. We have inserted code which will initialize an array of transport handles and for each role create an array of the procedures each role has access to. These will be read from the RPC IDL definition file and hardcoded into the server stub. Afterwards the code to generate the descriptive string is inserted. Most of the string is hard coded into the server stub, that is, the program and version name and number, and the role number is hardcoded into the string, so that the server stub only needs to insert the host name, the role name, and the current date. To obtain these parameters a call to `gethostname`, `get_role_name`, and `localtime` is inserted. `gethostname` and `localtime` is system calls, while `get_role_name` is a function we have added to the RPC library, which takes a role number, and reads the corresponding role name from the file generated by `rpc_rout` The name of the file which contains the public key is also hardcoded into the server stubs, since we know all the parameters to generate this from the RPC IDL definition file. After the code to generate the descriptive string and key file name is inserted, code which calls `svcudp_create` with these two and the role number as parameters is inserted. These three things are done for each role. When the code to create all the transport handles is generated, we iterate over the roles and insert a call to `svc_register` for each role, and at last the call to `svc_run` is inserted. After the main method is generated we iterate over all the program definitions given in the RPC IDL definition file, and generate a dispatch function for each program version.

rpc_kout.c contains the `generate_keys` function which takes a string and the length of the string as parameters and writes two files containing the data for the public and private keys. The string is used to generate the name of the files containing the keys. This is done by appending `.priv` and `.pub` to the string for the private and public key, respectively. `generate_keys` will first initialize the randomness generator and the key data structures by calling the Nettle functions: `yarrow256_init` and `simple_random` for the randomness generator and `rsa_public_key_init` and `rsa_private_key_init` for the keys. Now the keypair is created with a call to `rsa_generate_keypair`. The key data structures are transformed to s-expressions (arbitrarily nested lists, like the ones used in Lisp) with two calls to `rsa_keypair_to_sexp`. The first call transforms the parameters for the public key, and the second does it for both the keys. This is done because the server needs both the public and private key, since the public key is used to generate port number. Finally the two s-expressions are written to the files.

rpc_rout.c contains the `write_roles` function, which takes the input file and a string as input and writes for each version a file, that contains the role names and numbers for this version. The string is the path to where the file will be stored. `write_roles` is called from `r_output` declared in `rpc_main`, which calls it for each program definition, and passes the program definition and a path as parameters. The path is either the path where the input file is located, or an empty string if no path was given to the input file. `write_roles` first creates the name of the output file, which is on the form:

Program_<program number>_Version_<version number>_roles

For each role it will print one line in this file on the form:

role <role name> = <role number>

rpc_saout.c contains functions to write sample code for the server and the client. For the server sample code, it iterates over each procedure in each program version and writes and writes a method without any body. For the client sample code, it iterates over each program version and writes a function which will call all the procedure in the version. This is done by first inserting code to generate the descriptive string and name to the file in which the key is stored. Everything except the hostname, role name, role number, and the current date is hardcoded into the string, these are instead given as parameter to the function. After this code to generate the key file name is inserted, everything but the role number is hardcoded into the key file name. At last a call to `clnt_create` is inserted together with a call to the function in the client stub which will call the remote procedure. After these functions have been inserted a main method is inserted which will call each function for each role there is in the corresponding program version. Calls to `gethostname` and `localtime` is inserted to create the parameters to the functions, while the role numbers and role names are hardcoded into the calls.

6.6 Summary

In this chapter we have described the implementation of our prototype. The prototype is based on the design of a secure RPC system which uses cryptographic access control, as described in chapter 5. We have chosen to use the Nettle library in our implementation, we have given a description of this, and of the security concerns regarding cipher systems, key length, and hash algorithms.

The general structure of RPC and how we have incorporated access control into it is described together with how RPCgen generates code, and the changes we have made to RPCgen.

Our prototype lacks certain things which would be essential in a full implementation; The server should be able to change port numbers dynamically, the clients should be able to connect to a service even though a hash collision has occured on the server, and the systems should be able to communicate over other transport protocols than UDP, especially TCP. These things have not been implemented in our prototype due to time constraints, but they are not essential to the access control.

Chapter 7

Evaluation

For a full scale implementation of an RPC system, a test must be carried out on different levels, we will give a short description of three different test levels; Unit testing, integration testing, and functional testing. These only covers part of the in-house testing, which should be done before the release of a new version of an application. Besides this there exist many other test methods, e.g. end-user tests, performance test and stress test, just to mention a few. But these would normally be done by a test department, and should not be done by the developers of the system, who have too great a knowledge of the internal workings of the system to include the sequences the system is not prepared for into the test.

Unit test This is a test of each piece of code in the system, it requires that the tester has a knowledge of the internal program structure, since each function must be called in every possible setup and with all possibly combination of arguments, to ensure that all code statements, branches, paths, and conditions in the code are tested. It is very time consuming, and is usually done by the programmers

Integration test This test operates on a higher level, and is done by testing combined parts of an application to determine that they interface correctly

Functional test This is a test of a full setup of the system, and is most successful if it is done as black-box testing, that means, that the tester has no knowledge of the internal program structure

Since we have only implemented a prototype of CACRPC, and due time to time constraints and lack of resources, we will not do a full test. We assume that the original RPC implementation and the Nettle library works as specified, and we will not test them further. This means that we will only test that the client and server are able to communicate, and that our access control works. During which the encryption/decryption of messages and the naming scheme used to generate port numbers also will be tested.

We have done some unit testing during development, but this has not been done systematically, and we cannot be sure that every branch of the code has been tested. Since we have not documented the tests done during development, they will not be described further. Our test is divided into a test of the RPCgen library and a test of the RPC library, where the latter again is split up into a test of the server and a test of the client. For the server we test that it can start under different circumstances, and for the client we test the communication with the server in different setups to cover all the different aspects of the access control. Since it is very difficult to test the client without doing any actual communication with the server, the client side test is also a functional test of the entire system. At last we will do a performance test, and compare our results with results from the implementation or RPC which is included in GLIBC. Since we have done all the testing our self, it is of course insufficient.

7.1 Test

This section will document the tests carried out on our prototype implementation. The RPCgen code generator has been functionally tested by providing a series of different input programs and combinations of command line parameters. The RPC library has been tested by linking it against the generated stubs and programs (or slightly modified versions in some cases). The resulting client and server programs have been run, and it has been verified that their communications are carried out as expected. This includes inspection of the contents of transmitted network packets.

7.1.1 RPCgen

The test cases in table 7.1 cover each of the command line parameters for RPCgen. The input file is in all cases the simple RPC IDL program listed in appendix C.1. The results demonstrate that RPCgen correctly generates the various files containing stubs, sample code, keys, role numbers, etc.

7.2 RPC Library

We now test the functionality of the RPC library by linking programs generated by RPCgen against it, and testing that they behave correctly when run. In order to do this, minor modifications to the sample client and server code are introduced, e.g. filling in the procedure bodies on the server, and making the client output the call results.

7.2.1 Server Side

We have performed the tests listed below. The tests verify that a service can register and start under different circumstances.

1. Start server with simple service. This test will start the server with a program with one version, one procedure, and one role. We will verify that the server starts properly

2. Start server with complicated service. This test is the same as 1, but we have added an extra version to the program, which has two procedures and two roles, where the first role is the same, as in version 1, and the second role is allowed to call both procedures

3. Hash collision. This test will start two services, which will yield a hash collision. This is done by using the same descriptive string and key in a role in both services. We will verify that both services start properly, with one of them starting as normal, and the other performing the hashing algorithm twice due to the first port number being unavailable

All the tests were carried out with the expected results.

No	Input	Params.	Output file(s)	Comment	OK
1	test1.x	-c	test1_xdr.c	XDR filter generation, see appendix C.2	√
2	test1.x	-h	test1.h	Header generation, see appendix C.3	√
3	test1.x	-l	test1_clnt.c	Client stub generation, see appendix C.4	√
4	test1.x	-s udp	test1_svc.c	Server stub generation, see appendix C.5	√
5	test1.x	-k	Key files	4 files are generated: A public and private key for each of the two roles in the input program	√
6	test1.x	-r	Role file	See appendix C.6	√
7	test1.x	none	All of the above	Generates all of the above files	√
8	test1.x	-a	test1_server.c test1_client.c Makefile.test1 and all of the above	Generates all of the above files, as well as client and server sample code and a sample make file - the latter three are listed in appendices C.7, C.8, and C.9	√

Table 7.1: RPCgen test cases

7.2.2 Client Side

To all of the test it is a precondition that the service corresponding to the client is running on the server.

1. Call remote procedure. This test will call a remote procedure. We verify that the `add` procedure in the program used for testing RPCgen can correctly add two integer on the server and return the sum to the client

2. Call remote procedure with wrong key. This test will call a remote procedure on a port where the procedure is listening, but will encrypt the request with a key, which is not allowed access. We will verify that the client will not receive any reply, and that the server will generate a decryption error and discard the message

3. Call remote procedure on wrong port. This test will try to call a remote procedure on a port, where no service is listening. This is done by modifying the descriptive string in the client. We will verify that the client will not receive any reply, and will generate a timeout error

4. Connect with two client. This test will try to call a procedure from two clients, which both are members of the same role. We will verify that both clients receive a reply

5. Call procedure which is not accessible. This test will try to call a procedure, which the role does not have access to. This will be done by sending a request with a procedure number, which is not accessible for the role which the client is a member of. We verify that the client receives no reply, while the server prints an access control error and discards the message

All the tests were carried out with the expected results.

7.3 Performance

In order to evaluate how our implementation performs when compared to a standard RPC library, we have carried out a simple performance test. The added cryptographic operations obviously introduce some amount of overhead, but it is important that the difference in performance is not so great that our library becomes impractical to use. In order to time the execution of an RPC call, we create a simple program containing a procedure which has an array of integers as its argument. We compile several versions of the program, using arrays of differing sizes, and the execution time from when the client invokes the RPC call until it returns, are measured using the `gettimeofday` system call. Due to a limitation on UDP packet size imposed by RPC, we are unable to send a very large amount of traffic. In fact, no measurable difference is found between an array of a few elements, and one of 2000 elements, which is near the maximum that can be contained in a packet. The test shows our implementation to be about 80 times slower than the GLIBC implementation when running over a local virtual loopback network interface. Our implementation takes about 10 ms to send a message and receive a reply, whereas GLIBC does it in about 120 μs. In practice, communication over e.g. a 100Mbps network link would reduce the difference greatly. For instance, the theoretical time to transfer $8KB$ over such a link is 640 μs (disregarding all communication overhead). Taking this into account, the difference between the two implementations is reduced to a factor of about 15. If network latency and even slower links, which are often all that is available when communicating over the internet, the difference becomes almost negligible.

In a full implementation supporting the TCP protocol the packet size restriction would not apply. For larger packet sizes, the execution time will not be as dominated by the initial setup time due to key generation and asymmetric encryption and decryption of the message header. The only operations introduced that have execution times proportional to the packet size are the symmetric algorithm and the hash function, which should be significantly faster than the asymmetric algorithm and randomness generator. In an improved implementation one may also choose to eliminate the need for repeated key generation. The test clearly indicates that this improvement will be worthwhile, since the startup time is so important, however, it remains to be tested whether the public key cryptography or the key generation dominates the startup time.

Another performance consideration is of course the CPU load incurred on the server due to serving many simultaneous requests that all require cryptographic operations. This has not been looked into, but may become a problem for the scalability of the system, so it should be investigated in the future.

7.4 Further Work

During the implementation of the prototype we have chosen to omit some of the functionality described in chapter 5, this section will describe the things we have omitted, and a solution to how they could be implemented.

- Our prototype does not support communication over the TCP transport protocol. The changes we have made to the UDP part of RPC should easily be ported to the TCP part

- If a hash collision occurs during the generation of port numbers on the server, the clients will not be able to communicate with the colliding service. This could be fixed by letting the client recalculate the port number in the same way as the server, if they do not receive an answer from the server

- The server is not able to dynamically change port numbers for services. To fix this the select call in *svc_run*, should be set to time out when a service needs to change port number. This could be done by letting the RPC programmer pass a list of timeouts and time intervals to *svc_run*. So the first list for each role specifies the time to the next port number change, and the second list specifies the time interval between port number changes for each role. *svc_run* will then for each loop be able to calculate the time to the next port number change, and pass this to the select call. *svc_run* should also generate a new port number for the service which caused the timeout, and register it with this port number

- The use of timestamps to verify that the message is not a replay attack is not scalable for a large distributed system, since it is expensive to synchronize the clock of all the clients. Instead a nonce should be used. The first time a client sends a request to a particular server is should generate a random number, and include this in the message, for all subsequent requests it must increment this number (this is a slight simplification since we have to take into account, that UDP packets may arrive out of order). The server will need to keep the last nonces seen for each client, and upon reception of request it will check that the nonce is higher than the one sent in the previous request, or add a new entry to the list, if it is the first request from this client

- Performance could be increased by letting the clients reuse their session key. As it is now, the session key is used to verify the integrity of the second part of a request. This could be fixed by including the nonce in the digest of the second part of the request. An attacker would then need to know the nonce to change the second part

Chapter 8

Conclusion

In this project, we have investigated the possibility of incorporating cryptographic access control into remote procedure call. The main purpose was to develop a version of this protocol, where confidentiality and integrity of all data are ensured, and access to services can be administered at the procedure level by distributing keys to the authorized parties.

The result is a prototype implementation based on the original Sun Microsystems RPC 4.0 library. In our implementation, the RPC programmer is able to manage access to procedures by declaring roles. Each role has access to one or more procedures, and has an asymmetric key pair associated with it. By distributing this key to a user, he becomes a member of the role, and thus has access to the relevant procedures. The name *role* has been chosen because the developed model is reminiscent of the RBAC model. It should be noted that some requirements of RBAC are not fulfilled by our system, however. Our system does support traditional DAC schemes.

The changes from the original RPC library were designed to be as transparent to the programmer as possible. The transition was facilitated by the inclusion of an updated version of the RPCgen code generator, which partially automates the task of writing programs that use our library.

Additional security considerations were proposed, but not fully implemented. Instead of relying on a portmapper, our RPC protocol generates port numbers based on descriptive strings, naming the procedure to be accessed, and the cryptographic key. The port number is calculated using a cryptographic hash function. This approach serves to make reconnaissance attacks against the system much harder to perform.

The performance of the system was shown to be significantly worse than the original library. This was to be expected when introducing cryptographic operations, particularly asymmetric ones. We give several suggestions for improving the performance through changes to the protocol. However, the current implementation should be sufficiently fast for most usage scenarios. Our testing indicates that the implementation works as intended, but more thorough testing should be performed in order to validate it further.

Appendix A

Extended IDL Syntax

The following is a EBNF specification of part of our extended RPC IDL. Specifically, we describe the syntax related to the *program* keyword. The remainder of the specification is standard XDR, as defined by [1]. Note that the non-terminal <type-specifier> is defined by XDR. <text> and <nat> denote string and natural number literals, respectively.

```
<program> ::=
  "program" <program-name> "{"
    <version-list>
  "} =" <program-no> ";"

<program-name> ::= <text>

<program-no> ::= <nat>

<version-list> ::= <version>+

<version> ::=
  "version" <version-name> "{"
    <proc-list>
    <role-list>
  "} = <version-no> ";"

<version-name> ::= <text>

<program-no> ::= <nat>

<proc-list> ::= <proc>+

<proc> ::=
  <type-specifier> <proc-name> "(" <argument> ") =" <proc-no> ";"

<proc-name> ::= <text>

<proc-no> ::= <nat>
```

```
<argument> ::= <type-specifier> <argument-name>

<argument-name> ::= <nat>

<role-list> ::= <role>+

<role> ::=
  "role" <role-name> "{" <proc-no-list> "} =" <role-no> ";"

<role-name> ::= <text>

<role-no> ::= <nat>

<proc-no-list> ::= <proc-no> ("," <proc-no>)*
```

Appendix B

RPCgen Manual

Our version of RPCgen is invoked from the command line, using the syntax described below. It is assumed that the user has previously created a file containing a definition of the program to be generated in RPC IDL. This file is usually named with a `.x` suffix, and the default names of most of the output files generated by RPCgen will be based on the name of the input file. The examples below assume that the input file is called `name.x`, such that e.g. the default filename for sample client code will be `name_client.c`. If input and output filenames are omitted, standard input and output will be used instead.

```
usage: /usr/bin/cacrpc/rpcgen [ -c | -h | -l | -m | -s udp]
                              [-o outfile] [infile]
       /usr/bin/cacrpc/rpcgen [-k | -r] [infile]
       /usr/bin/cacrpc/rpcgen [-a] infile
```

The individual options are explained below. They are divided into three groups. For the first group, only one operation is performed, so both input and output files are optional:

-c Generate XDR filter routines for converting to and from any XDR types defined in the input file

-h Generate header file containing definitions of constants such as program numbers

-l Generate client stub

-m Generate server stub without main function

-s udp Generate server stub with main function, using the UDP protocol.

The second group of options have several output files. The names of these cannot be specified.

-k Generate public and private keys for each role in the program. These are placed in files named after the program, with the suffix `.key`

-r Generate a file containing a list of role numbers and their names, as found in the program. This file is used at runtime by the client and server to map numbers into names. The file is named similarly to key files, but with suffix `.roles`

Finally, the last two options allow us to perform a lot of tasks at once - for this reason the input filename must be specified, and the output filenames are automatically generated based on the input file name:

no options Generate the following files:

`name.h` Header file as generated using -h

`name_clnt.c` Client stub as generated using -l

`name_svc.c` Server stub as generated using -s udp

`name_xdr.c` XDR filters as generated using -c

- Key files as generated using -k

- Role file as generated using -r

-a This option generates all the files above, and in addition creates the following sample files:

`name_client.c` Sample client code

`name_server.c` Sample server code

`Makefile.name` Sample make file for compiling the RPC program

Appendix C

Test Input and Results

The files below are input and output from testing RPCgen. See section 7.1.

C.1 test1.x

```
struct INT_PAIR {
  int A;
  int B;
};

program TEST1_PRG {
  version TEST1_VER {
    int add ( INT_PAIR ) = 1;
    INT_PAIR divide ( INT_PAIR ) = 2;
    role ROLE1 {1} = 1;
    role ROLE2 {1,2} = 2;
  } = 1;
} = 0x40000001;
```

C.2 test1_xdr.c

```
#include <cacrpc/rpc.h>
#include "test1.h"


bool_t
xdr_INT_PAIR(xdrs, objp)
  XDR *xdrs;
  INT_PAIR *objp;
{
  if (!xdr_int(xdrs, &objp->A)) {
    return (FALSE);
  }
  if (!xdr_int(xdrs, &objp->B)) {
    return (FALSE);
```

```
  }
  return (TRUE);
}
```

C.3 test1.h

```
#include <cacrpc/rpc.h>

struct INT_PAIR {
  int A;
  int B;
};
typedef struct INT_PAIR INT_PAIR;
bool_t xdr_INT_PAIR();


#define TEST1_PRG ((u_long)0x40000001)
#define TEST1_VER ((u_long)1)
#define add ((u_long)1)
extern int *add_1();
#define divide ((u_long)2)
extern INT_PAIR *divide_1();

#define NO_OF_ROLES 2
```

C.4 test1_clnt.c

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "test1.h"
#include<time.h>

void
test1_prg_1(host, role_num, role_name, date_now)
  char *host;
  char *role_num;
  char *role_name;
  char *date_now;
{
  CLIENT *clnt;
  size_t size;
  char *hash_string;
  char *keyfile;
  int  *result_1;
```

```
  INT_PAIR  add_1_arg;
  INT_PAIR  *result_2;
  INT_PAIR  divide_1_arg;

  size = strlen(host) + 8 + 10 + 9 + 1 + 1 + 9 + 1 + 9 + 1 +
strlen(role_name) + 1 + strlen(role_num) + 7 + 1;
  hash_string = (char *) calloc(size, sizeof(char));
  snprintf(hash_string, size,
"%s/Program_0x40000001_Version_1/TEST1_PRG/TEST1_VER/%s/%s/%s", host,
role_name, role_num, date_now);
  size = 8 + 10 + 9 + 1 + 6 + strlen(role_num) + 7 + 1;
  keyfile = (char *) calloc(size, sizeof(char));
  snprintf(keyfile, size, "Program_0x40000001_Version_1_Role_%spub.key",
role_num);
  clnt = clnt_create(host, hash_string, keyfile,"udp");
  if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
  }

  result_1 = add_1(&add_1_arg, clnt);
  if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
  }
  result_2 = divide_1(&divide_1_arg, clnt);
  if (result_2 == (INT_PAIR *) NULL) {
    clnt_perror (clnt, "call failed");
  }
  free(hash_string);
  free(keyfile);

  clnt_destroy (clnt);
}

int
main(argc, argv)
  int argc;
  char *argv[];
{
  char *host;
  struct tm *time_now;
  time_t secs_now;
  char date_now[8];

  if (argc != 2) {
    fprintf(stderr, "usage: %s <hostname>\n",argv[0])
    exit(1);
  }
  host = argv[1];

  time(&secs_now);
  time_now = localtime(&secs_now);
  strftime(date_now,7,"%m/%y",time_now);
  test1_prg_1 (host, "1", "ROLE1", date_now);
```

```
  test1_prg_1 (host, "2", "ROLE2", date_now);
  exit(0);
}
```

C.5 test1_svc.c

```
#include <stdio.h>
#include <unistd.h>
#include <cacrpc/rpc.h>
#include <time.h>
#include "test1.h"

static void test1_prg_1();

main()
{
  SVCXPRT *transp[NO_OF_ROLES];
  u_long roleTEST1_PRG11[1] ={1};
  u_long roleTEST1_PRG12[2] ={1,2};
  int role_idx;
  char *host_name;
  char *role_name;
  size_t size;
  char *name_string;
  char *keyfile;
  struct tm *time_now;
  time_t secs_now;
  char date_now[8];


  host_name = (char *) calloc(30, sizeof(char));
  gethostname(host_name, 30);
  time(&secs_now);
  time_now = localtime(&secs_now);
  strftime(date_now,7,"%m/%y",time_now);

  if (!get_role_name(&role_name, "Program_0x40000001_Version_1_roles",
1)) {
    (void)fprintf(stderr, "cannot get identifier for role 1 program
0x40000001 version 1.\n");
    exit(1);
  }
  size = strlen(host_name) + 1 + 28 + 1 + 9 + 1 + 9 + 1 +
strlen(role_name) + 1 + 1 + 1 + strlen(date_now) + 1;
  name_string = (char *) calloc(size, sizeof(char));
  snprintf(name_string, size,
"%s/Program_0x40000001_Version_1/TEST1_PRG/TEST1_VER/%s/1/%s", host_name,
role_name, date_now);
  size = 8 + 10 + 9 + 1 + 6 + 1 + 8 + 1;
  keyfile = (char *) calloc(size, sizeof(char));
  snprintf(keyfile, size, "Program_0x40000001_Version_1_Role_1priv.key");
  transp[0] = svcudp_create(name_string, keyfile, 1);
```

```
  if (transp[0] == NULL) {
    (void)fprintf(stderr, "cannot create udp service.\n");
    exit(1);
  }
  if (!get_role_name(&role_name, "Program_0x40000001_Version_1_roles",
2)) {
    (void)fprintf(stderr, "cannot get identifier for role 2 program
0x40000001 version 1.\n");
    exit(1);
  }
  size = strlen(host_name) + 1 + 28 + 1 + 9 + 1 + 9 + 1 +
strlen(role_name) + 1 + 1 + 1 + strlen(date_now) + 1;
  name_string = (char *) calloc(size, sizeof(char));
  snprintf(name_string, size,
"%s/Program_0x40000001_Version_1/TEST1_PRG/TEST1_VER/%s/2/%s", host_name,
role_name, date_now);
  size = 8 + 10 + 9 + 1 + 6 + 1 + 8 + 1;
  keyfile = (char *) calloc(size, sizeof(char));
  snprintf(keyfile, size, "Program_0x40000001_Version_1_Role_2priv.key");
  transp[1] = svcudp_create(name_string, keyfile, 2);
  if (transp[1] == NULL) {
    (void)fprintf(stderr, "cannot create udp service.\n");
    exit(1);
  }

  if (!svc_register(transp[0], &roleTEST1_PRG11, 1, test1_prg_1,
IPPROTO_UDP)) {
    (void)fprintf(stderr, "unable to register (TEST1_PRG, TEST1_VER,
udp).\n");
    exit(1);
  }
  if (!svc_register(transp[1], &roleTEST1_PRG12, 2, test1_prg_1,
IPPROTO_UDP)) {
    (void)fprintf(stderr, "unable to register (TEST1_PRG, TEST1_VER,
udp).\n");
    exit(1);
  }
  svc_run();
  (void)fprintf(stderr, "svc_run returned\n");
  exit(1);
  free(host_name);
  free(keyfile);
}

static void
test1_prg_1(rqstp, transp)
  struct svc_req *rqstp;
  SVCXPRT *transp;
{
  union {
    INT_PAIR add_1_arg;
    INT_PAIR divide_1_arg;
  } argument;
  char *result;
```

```
  bool_t (*xdr_argument)(), (*xdr_result)();
  char *(*local)();

  switch (rqstp->rq_proc) {
  case NULLPROC:
    (void)svc_sendreply(transp, xdr_void, (char *)NULL);
    return;

  case add:
  {
    xdr_argument = xdr_INT_PAIR;
    xdr_result = xdr_int;
    local = (char *(*)()) add_1;
    }
    break;
  case divide:
  {
    xdr_argument = xdr_INT_PAIR;
    xdr_result = xdr_INT_PAIR;
    local = (char *(*)()) divide_1;
    }
    break;
  default:
    svcerr_noproc(transp);
    return;
  }
  bzero((char *)&argument, sizeof(argument));
  if (!svc_getargs(transp, xdr_argument, &argument)) {
    svcerr_decode(transp);
    return;
  }
  result = (*local)(&argument, rqstp);
  if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
    svcerr_systemerr(transp);
  }
  if (!svc_freeargs(transp, xdr_argument, &argument)) {
    (void)fprintf(stderr, "unable to free arguments\n");
    exit(1);
  }
}
```

C.6 Program_0x40000001_Version_1_roles

```
role ROLE1 = 1
role ROLE2 = 2
```

C.7 test1_server.c

```
/*
```

```
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "test1.h"

int *
add_1(INT_PAIR *argp, struct svc_req *rqstp)
{
  static int result;

  /*
   * insert server code here
   */

  return &result;
}

INT_PAIR *
divide_1(INT_PAIR *argp, struct svc_req *rqstp)
{
  static INT_PAIR result;

  /*
   * insert server code here
   */

  return &result;
}
```

C.8 test1_client.c

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "test1.h"
#include<time.h>

void
test1_prg_1(host, role_num, role_name, date_now)
  char *host;
  char *role_num;
  char *role_name;
  char *date_now;
{
  CLIENT *clnt;
  size_t size;
  char *hash_string;
```

```
  char *keyfile;
  int  *result_1;
  INT_PAIR  add_1_arg;
  INT_PAIR  *result_2;
  INT_PAIR  divide_1_arg;

  size = strlen(host) + 8 + 10 + 9 + 1 + 1 + 9 + 1 + 9 + 1 +
strlen(role_name) + 1 + strlen(role_num) + 7 + 1;
  hash_string = (char *) calloc(size, sizeof(char));
  snprintf(hash_string, size,
"%s/Program_0x40000001_Version_1/TEST1_PRG/TEST1_VER/%s/%s/%s", host,
role_name, role_num, date_now);
  size = 8 + 10 + 9 + 1 + 6 + strlen(role_num) + 7 + 1;
  keyfile = (char *) calloc(size, sizeof(char));
  snprintf(keyfile, size, "Program_0x40000001_Version_1_Role_%spub.key",
role_num);
  clnt = clnt_create(host, hash_string, keyfile,"udp");
  if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
  }

  result_1 = add_1(&add_1_arg, clnt);
  if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
  }
  result_2 = divide_1(&divide_1_arg, clnt);
  if (result_2 == (INT_PAIR *) NULL) {
    clnt_perror (clnt, "call failed");
  }
  free(hash_string);
  free(keyfile);

  clnt_destroy (clnt);
}

int
main(argc, argv)
  int argc;
  char *argv[];
{
  char *host;
  struct tm *time_now;
  time_t secs_now;
  char date_now[8];

  if (argc != 2) {
    fprintf(stderr, "usage: %s <hostname>\n",argv[0])
    exit(1);
  }
  host = argv[1];

  time(&secs_now);
  time_now = localtime(&secs_now);
```

```
  strftime(date_now,7,"%m/%y",time_now);
  test1_prg_1 (host, "1", "ROLE1", date_now);
  test1_prg_1 (host, "2", "ROLE2", date_now);
  exit(0);
}
```

C.9 Makefile.test1

```
# This is a template Makefile generated by rpcgen

# Parameters

CLIENT = test1_client
SERVER = test1_server

SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = test1.x

TARGETS_SVC.c = test1_svc.c test1_server.c test1_xdr.c
TARGETS_CLNT.c = test1_clnt.c test1_client.c test1_xdr.c
TARGETS = test1.h test1_xdr.c test1_clnt.c test1_svc.c test1_client.c
test1_server.c

OBJECTS_CLNT = $(SOURCES_CLNT.c:%.c=%.o) $(TARGETS_CLNT.c:%.c=%.o)
OBJECTS_SVC = $(SOURCES_SVC.c:%.c=%.o) $(TARGETS_SVC.c:%.c=%.o)
# Compiler flags

CFLAGS += -g -L/usr/lib/cacrpc
LDLIBS += -lrpclib -lnettle -lgmp
RPCGENFLAGS =

# Targets

all : $(CLIENT) $(SERVER)

$(TARGETS) : $(SOURCES.x)
  rpcgen $(RPCGENFLAGS) $(SOURCES.x)

$(OBJECTS_CLNT) : $(SOURCES_CLNT.c) $(SOURCES_CLNT.h) $(TARGETS_CLNT.c)

$(OBJECTS_SVC) : $(SOURCES_SVC.c) $(SOURCES_SVC.h) $(TARGETS_SVC.c)

$(CLIENT) : $(OBJECTS_CLNT)
  $(LINK.c) -o $(CLIENT) $(OBJECTS_CLNT) $(LDLIBS)

$(SERVER) : $(OBJECTS_SVC)
  $(LINK.c) -o $(SERVER) $(OBJECTS_SVC) $(LDLIBS)

 clean:
```

```
   $(RM) core $(TARGETS) $(OBJECTS_CLNT) $(OBJECTS_SVC) $(CLIENT)
$(SERVER)
```

Bibliography

[1] R. Srinivasan. XDR: External data representation standard, August 1995. RFC 1832.

[2] R. Srinivasan. Binding protocols for ONC RPC version 2, August 1995. RFC 1833.

[3] John Shapley Gray. *Interprocess Communications in LINUX: The Nooks and Crannies.* Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 2003.

[4] Digital UNIX. Programming with ONC RPC. `http://www.cs.arizona.edu/computer.help/policy/DIGITAL_unix/AA-Q0R5B-TE%T1_html/TITLE.html`, March 1996.

[5] Greg O'Shea and Michael Roe. Child-proof authentication for MIPv6 (CAM). *CCR*, Apr(31), 2001.

[6] T. Aura. Cryptographically generated addresses (CGA), April 2004. draft-ietf-send-cga-06.

[7] Jari Arkko, Tuomas Aura, James Kempf, Vesa-Matti Mäntylä, Pekka Nikander, and Michael Roe. Securing IPv6 neighbor and router discovery. In *Proceedings of the ACM Workshop on Wireless Security (WiSe-02)*, pages 77–86, New York, September 28 2002. ACM Press.

[8] J. Arkko, J. Kempf, B.Sommerfeld, B. Zill, and P. Nikander. Secure neighbor discovery (SEND), July 2004. draft-ieft-send-ndopt-06, work in progress.

[9] Claude Castelluccia. Statistically unique and cryptographically verifiable (SUCV) identifiers and addresses. November 23 2001.

[10] Adi Shamir. Identity-based cryptosystem and signature scheme. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology – CRYPTO ' 84*, volume 196 of *Lecture Notes in Computer Science*, pages 120–126. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1984.

[11] Dan Boneh and Matthew Franklin. Identity-based encryption from the weil pairing. *Lecture Notes in Computer Science*, pages 213–229, February 12 2001.

[12] J. Kempf, C. Gentry, and A. Silverberg. Securing IPv6 neighbor discovery using address based keys (ABKs), May 2003. draft-kempf-abk-nd-00.txt, work in progress.

[13] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramoli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[14] Anthony Harrington and Christian Jensen. Cryptographic access control in a distributed file system. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT-03)*, pages 158–168, New York, June 2–3 2003. ACM Press.

[15] S. Hjarlvig and J. Kampfeldt. Kryptografisk adgangskontrol i peer-to-peer netværk. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2003. Vejleder: Christian D. Jensen.

[16] Charles Pfleeger and Shari Lawrence Pfleeger. *Security in computing*. Prentice Hall International, 3rd edition edition, 2003.

[17] Niels Möller. Nettle - a low-level cryptographic library. `http://www.lysator.liu.se/~nisse/nettle/`.