Charon: Kerberos^{TM} Extensions For Authentication Over Secondary Networks

by

Derek A. Atkins

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Electrical Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

Author ....
Department of Electrical Engineering and Computer Science
May 7, 1993

Certified by ....
Jeffrey I. Schiller
Network Manager
Thesis Supervisor

Accepted by ....
Leonard A. Gould
Chairman, Departmental Committee on Graduate Students

Charon: Kerberos^{TM} Extensions For Authentication Over Secondary Networks

by

Derek A. Atkins

Submitted to the Department of Electrical Engineering and Computer Science
on May 7, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Electrical Science and Engineering

Abstract

In this thesis, I describe extensions to the Kerberos Authentication System to enable a secure method of Authentication over multiple networks. Kerberos was designed with a fully-connected IP network in mind, however when you add dialup capabilities to the picture, Kerberos doesn't expand to secure the whole connection.

Charon was created to tackle this problem. It was developed to provide a way to securely authenticate to a login server over a modem connection, without allowing a passive attacker to gain enough information to impersonate the user. This means that a user can log into a Kerberized host without typing his password in clear-text over the phone. In addition, no modifications to the login server's base operating system need to be made in order to accomplish this.

Thesis Supervisor: Jeffrey I. Schiller
Title: Network Manager

---

^{TM}Kerberos is a trademark of the Massachusetts Institute of Technology

Charon: Kerberos^{TM} Extensions For Authentication Over Secondary Networks

by

Derek A. Atkins

Submitted to the Department of Electrical Engineering and Computer Science
on May 7, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Electrical Science and Engineering

Abstract

In this thesis, I describe extensions to the Kerberos Authentication System to enable a secure method of Authentication over multiple networks. Kerberos was designed with a fully-connected IP network in mind, however when you add dialup capabilities to the picture, Kerberos doesn't expand to secure the whole connection.

Charon was created to tackle this problem. It was developed to provide a way to securely authenticate to a login server over a modem connection, without allowing a passive attacker to gain enough information to impersonate the user. This means that a user can log into a Kerberized host without typing his password in clear-text over the phone. In addition, no modifications to the login server's base operating system need to be made in order to accomplish this.

Thesis Supervisor: Jeffrey I. Schiller
Title: Network Manager

---
^{TM}Kerberos is a trademark of the Massachusetts Institute of Technology

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer network security is inherently a problem of trust: in order to believe that a user is who he claims, you need to have some method to trust that user. At MIT/Athena, the Kerberos Authentication System ([6]) was developed as an authentication system. However, the MIT implementation of Kerberos was designed to work on a fully-connected IP network; it was not designed to work over multiple types of networks, such as a combination of LAN and phone-modem. This thesis describes Charon, created to secure secondary networks and provide Kerberos authentication over previously insecure networks.

The name Charon is derived from Greek mythology, where he was the ferryman to souls over the river Styx to Hades. The myth continues in that Charon requires a passenger to verify their identity with some token before he would ferry them safely across. Similarly, the Charon protocol expects a user to present a password as an identifying token before it safely will pass a Kerberos ticket across to from the client to login server. This will be described in more detail later.

The rest of this chapter discusses the status quo.

Chapter two describes the Charon security goals, protocol design, and assumptions.

Chapter three describes the implementation of the Charon library.

Chapter four describes the Charon API and applications.

Chapter five describes the testing that was done on the system.

Chapter six contains conclusions and possible future enhancements to the Charon system.

1.1 Old security methods

In trying to secure a computer network, one first has to make some assumptions about the network to be secured. In the first security model, users had to type their password over the network *in the clear* in order to log into a remote host. This means that anyone listening on the network can receive their transmissions and grab the password. An attacker can then retransmit this password at a future time to gain access to the user's account.

The next step for security, on most UNIX^{TM} systems, is accomplished through the concept of trusted hosts and trusted users. What this means is that attempting to log on to a machine from a trusted host as a trusted user will be accepted by the machine without typing a password. In other words, a "trusted" machine, asserting a user's identity, is assumed to be honest.

This is done, in the Berkeley UNIX style, by using hosts.equiv and .rhosts files. These files tell a machine which hosts it trusts, and which host/user pairs a user trusts. Therefore, if a user wants to be able to login to one host, A, from another, B, then they need to either have B in A's hosts.equiv file, or have B in their own .rhosts file.

However this type of "security", through machine trust, can cause problems. Since anyone who can become root on a host can then become any other user via the su command, they can then wander the network as this other user, following a path of machines that trust that host or account. In other words, not all machines can be assumed to be honest. Another problem of the old security methods is that if a user is not on a trusted machine then they still have to type their password in the clear across the network, which reverts to the old problem. Another problem is that if an attacker can spoof the name of her host to be that of a trusted host, they can again gain unauthorized access to other machines across the network.

1.2 The Kerberos model

These problems were addressed with the advent of Kerberos by MIT Project Athena. Kerberos is a network authentication system that solves many of the problems of standard UNIX security by removing the concept of a trusted host, and changing the concept of a trusted user. The concept of Kerberos is that everyone and everything, users, hosts, services, etc., has a name and a secret key. The only entities that know these secrets are the owner (i.e., the user or the service-providing machine) and the Kerberos Server. In other words, the key is a "shared secret" between the named entity and the Kerberos Server [2, 5].

The Kerberos Server, also called the Key Distribution Center (or KDC), has the job of issuing tickets to users. A ticket is a cryptographic block that a user can present to a service provider to prove the user's identity. The authentication begins by the user telling the KDC their name. The KDC then sends the user a Ticket Granting Ticket (TGT), encrypted in the user's key. The user then types their password on their workstation. The workstation translates this password into the user's key, which is used to decrypt the TGT. The workstation can use the TGT to obtain other tickets for any services the user may wish to use.

If the user wants to log into a remote machine, for example, all she needs to do is obtain a TGT,

^{TM}UNIX is a trademark of Unix System Laboratories, Novell, or some such company which is secure because she is only typing her password to her local machine. The obtained TGT is used to request a service ticket for the host to which she wants to login. The service ticket is a cryptographic block, like the TGT, which is encrypted in the shared key between the service's host and the KDC. The client then passes this service ticket to the login server and, if the credentials are valid, the user will get logged in without having to type anything in clear-text over the network. In addition, the user can optionally encrypt this session so that all data sent over the network between the remote host and their local host is encrypted.

Kerberos makes a number of assumptions of the machines and network it wants to secure[1]. One assumption is that all decryption of tickets is done locally on the client machine. This means that the client must have enough computing power to do a Data Encryption Standard (DES) [4] decryption in a reasonable amount of time. Another assumption in the MIT implementation of Kerberos is that a client must be able to speak directly to the Kerberos Server in order to obtain tickets. One reason for this is that the KDC uses the IP address of the client machine as part of the Kerberos ticket.

1.3 The Problem

However, now assume that there are two separate networks which are connected only though some intermediate host, for example a phone modem connected to a machine on the Internet. In this case I call the phone line and modem the "secondary network". In this setup, using standard Kerberos, there is no way to secure the connection between the client machine and the server machine on either end of the modem connection.[2] In this situation, given the standard Kerberos protocol, it is possible for an attacker to get enough information to actually log in as the user if the modem connection is monitored. An example of this network is in figure 1-1, which shows a client terminal, connected via modem to a server machine, which is on an ethernet with the KDC.

In this network, assuming that the server host is using Kerberos, an attacker can look at the secondary network, which is insecure, and grab the password or any data that crosses that line. In addition, she could also watch the primary network, the ethernet in this example, and grab the Kerberos packets. Then, using the user's password, she can become that user and decode all the user's data.[3] What's even worse is that this attacker now has enough information (the password) to impersonate that user on the network. The bad guy just needs to try to log into the machine and present the user's password (which was conveniently grabbed earlier) at the prompt.

To protect against this kind of attack, it would be nice to extend the Kerberos protocol across the modem link in order to secure that section of the connection, so that a user can log into a host server

[1] see [6] for more details

[2] It is possible to run some sort of IP layer over the modem, such as SLIP, which would enable the client to talk directly to the Kerberos Server. If IP connectivity is available, then Charon would not be necessary. However, I am trying to deal with the case in which it is not possible to run such an environment.

[3] This attack requires that *all* the user's data (Kerberos packets in particular) has been captured

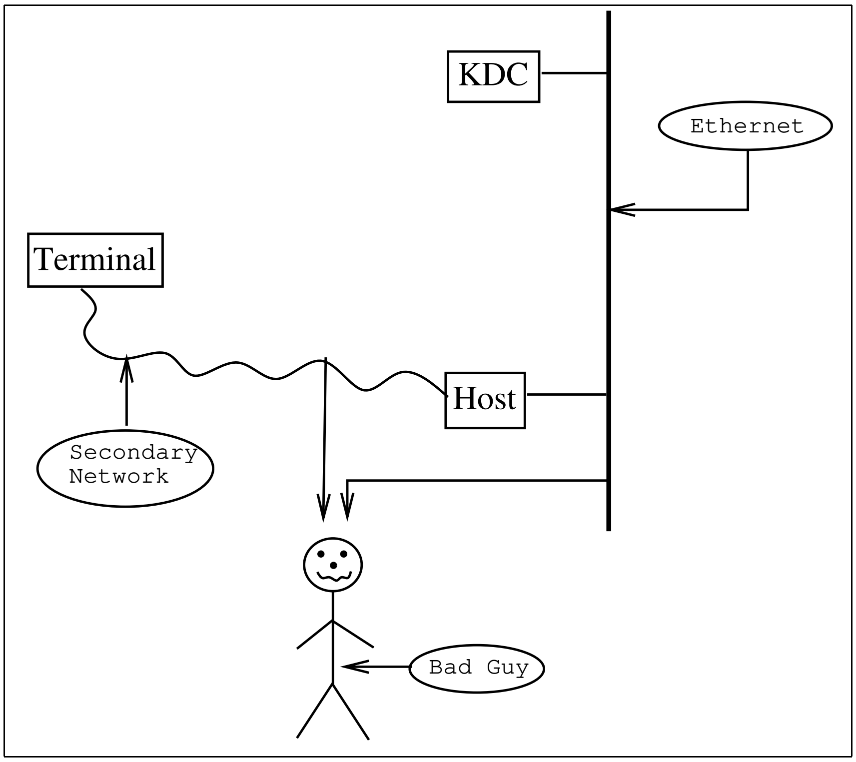

Figure 1-1: Network Topology. Note that the bad guy has access to both the primary (ethernet) and secondary (phone) networks.

from a client terminal without sending enough information for an attacker to use to gain access as that user. This system, described in the rest of this paper, is the Charon Protocol extensions to Kerberos over secondary networks.

Chapter 2

Design Goals of the Charon System

The first assumption Charon makes that that there can be two separate networks between the client machine and the KDC, connected by a host which I will call the *login server.* These two networks, the primary network and secondary network, communication with each other only through the login server. A "Secondary Network," as I use the term, would include the phone network, if it connects via modem a terminal client and a login server, and the login server is on the Internet. We have to make a few assumptions of the situation which we are attempting to secure:

- Both networks are insecure. This means that an eavesdropper can listen to the network, as well as the modem connection, and read anything that is sent in clear text.
- The terminal does not have a way direct to communicate with the KDC to obtain Kerberos credentials.
- The terminal to host link may be slow[1] and/or single threaded, like a serial connection or a modem.

2.1 Security Goals

The goals of this design is the create a protocol that is as secure as the standard Kerberos protocol. It should be quick, versatile, and simple to use and integrate into many applications. It should not be vulnerable to replay attacks. The security goals of this design are:

- To authenticate the user on the terminal to the login server using Kerberos.

[1] slow compared to a typical LAN speed

- To deposit a Kerberos TGT on the login server after authentication.
- Passwords, and any other information that would give a passive attacker an advantage, never cross the net, either primary or secondary, in clear text.
- To give the user the option to encrypt the session.
- To keep an attacker from getting enough information to become the user. This means that it should be secure against replay attacks, the attacker shouldn't be able to obtain the users' credentials, or trick the host using this protocol.

The protocol should defend against the same attacks that Kerberos does. For example, passive eavesdropping should not yield any information. It should guard against replay attacks, so that an eavesdropper can't try to use someone else's data and log in as them. Such an attack is comprised of an attacker monitoring the modem line, recording the transmissions from the client to the login server, and then replaying those transmissions as their own, in hope that the login server will believe that the attacker is who he says he is. It should also be safe against spoofing from the server host as well, so that it can't trick the user into giving away their tickets to an attacker.

The protocol should also allow for optional encryption of the session after the Kerberos exchange takes place. Possibilities are no encryption, encrypting from client to server only, encrypting from server to client only, or two-way encryption.

It should also allow for the ability to re-authenticate, so that once connected the user can get another Kerberos ticket. This means that the protocol should be usable from a login shell, and shouldn't require any kind of out-of-band connections. In other words, there needs to be some kind of program that runs on the server that should be self contained, and should use the terminal for input and output. This method was chosen so that no modifications to the login server's operating system are necessary.

Moreover, users should be able to decrypt the Kerberos information offline, in case their terminal program doesn't understand the Charon protocol. For example, a user should be able to capture the cryptographic information into a file, and then escape into another program which decrypts that file and creates a response, which can then be shipped across to the host.

2.2 Protocol Goals

The protocol design makes the following assumptions:

- Both networks can send at least six-bit data, which includes the IA5 subset of the ASCII character set.

- The terminal does not necessarily need to know about what kind of network the primary network is, and should get all relevant information it needs from the login server.

- The protocol should be extensible.

The protocol should work over connections that are not 8-bit clean, meaning that it shouldn't depend upon being able to send 8-bit data, although it should send 8-bit data if the connection will allow it. I decided on a six bit protocol, which will allow data to traverse through protocol translators and other such devices or programs which might not pass some control sequences or other 8-bit data. This is a safety measure so that nothing in the data could possibly be an escape character in any piece of hardware along the link.

Since the client shouldn't need to know about the server network, it doesn't need to know about things like sockets, TCP/IP, hostnames or name resolution, Kerberos realms or servers, or anything else that a normal Kerberos client depends upon.

2.3 Secondary Kerberos Protocol

The secondary Kerberos protocol should be based, in part, on the normal Kerberos protocol. One possibility would be to just have the login server pass Kerberos packets from the secondary network onto the primary network, and just use the standard protocol as it was originally designed. While this solution works, its not very elegant, because it requires multiple round-trips for the data. Extra round trips can mean a longer delay (120 bytes will take one second to transfer on a 1200 baud modem). By making the protocol into a single series of exchanges, instead of the standard Kerberos exchange, it becomes possible to easily create an offline application program so you don't have to modify the communications program resident on the client computer.

Normally, to get a secure connection between a client and a server, the client would get a TGT, and then request a service ticket for the server. In other words, we require at least two tickets on the client.[2] It is this requirement of multiple tickets that would force the multiple round trips using the standard Kerberos protocol.

To reduce the number of round trips, we can request two tickets together, the TGT and a service ticket, and send both of them to the client at the same time. Then the user can decrypt them together. The client then can use the service ticket for a secure connection to the server host, and send the TGT. In order to make sure that it is really the user requesting the tickets, both the TGT and service tickets are encrypted in the secret key of the user, K_u. Graphically, this exchange is depicted in figure 2-1.

[2] In Kerberos version 4, we require exactly two or three tickets on the client, depending on whether the user principal and server are in the same realm or not.

We have to be careful with replay attacks. An attacker could watch the modem connection, grab the encrypted TGT, and then replay it to the server and gain access to the user's account. This is supposed to be prevented in Kerberos by remembering the authenticators, and the Internet (IP) Address of the host is a part of that identity.[3] However in a dialup situation, or on a secondary network, the client machine doesn't have an IP address for use in the authenticator. Therefore, we need some other form of replay protection, since Kerberos version 4 doesn't have a working replay cache.

The *cookie* in this protocol is the protection against replay attacks. It is a piece of data, guaranteed unique, which is created by the login server and used to prove the session identity. The server sends the cookie to the client with the initial tickets, and it expects the client to send the same cookie back to the server, encrypted in the session key from the service ticket. This means that it requires knowledge of the shared key between the client and server machines, which a passive attacker cannot know. To be unique, the *cookie* should be some function of the time and username, so that it is known to be unique. This implementation just uses the time, and makes sure it is usique by locking out any other process from grabbing a cookie until this one has it.

Another problem we have to watch is time skew. Since the terminal may not be synchronized to the host, the time of the client may not help prevent replays.

In the protocol exchanges, the term u refers to the user's name, tgs is the ticket granting service's name. The entity represented by ws is the login server. Also, I use K_u to represent the user's secret key, and $K_{x,y}$ to represent the shared key between the entities x and y.

Step	*Direction*	*Data*
1	Host $\Rightarrow$ KDC	u, tgs, ws
2	Host $\Leftarrow$ KDC	$\{T_{u,tgs}, K_{u,tgs}\}K_u$, $\{T_{u,ws}, K_{u,ws}\}K_u$
3	Terminal $\Leftarrow$ Host	$\{T_{u,tgs}, K_{u,tgs}\}K_u$, $\{T_{u,ws}, K_{u,ws}\}K_u$, $COOKIE$
4	Terminal $\Rightarrow$ Host	$T_{u,ws}, A_{u,ws}$, $\{T_{u,tgs}, K_{u,tgs}\}K_{u,ws}$, $\{COOKIE\}K_{u,ws}$

Ticket	$T_{p,s} = \{p, s, addr_{ws}, time, life, K_{p,s}\}K_s$
Authenticator	$A_{p,s} = \{p, addr_{ws}, time\}K_{p,s}$
$COOKIE$	A random value created by the host to protect against replays.

Figure 2-1: Proposed Protocol with multiple key-packets in a single transmission.

[3] Timestamps, replay caches, and/or challenge-response schemes are also in use.

Step	*Direction*				*Data*
1		Host	$\Rightarrow$	KDC	$u, tgs, rtgs$
2		Host	$\Leftarrow$	KDC	$\{T_{u,tgs}, K_{u,tgs}\}K_u$, $\{T_{u,rtgs}, K_{u,rtgs}\}K_u$
3	Terminal	$\Leftarrow$ Host			$\{T_{u,tgs}, K_{u,tgs}\}K_u$, $\{T_{u,rtgs}, K_{u,rtgs}\}K_u$
4	Terminal	$\Rightarrow$ Host			$u, ws, T_{u,rtgs}, A_{u,rtgs}$
5		Host	$\Rightarrow$	RKDC	$u, ws, T_{u,rtgs}, A_{u,rtgs}$
6		Host	$\Leftarrow$	RKDC	$\{T_{u,ws}, K_{u,ws}\}K_{u,rtgs}$
7	Terminal	$\Leftarrow$ Host			$\{T_{u,ws}, K_{u,ws}\}K_{u,rtgs}$, $COOKIE$
8	Terminal	$\Rightarrow$ Host			$T_{u,ws}, A_{u,ws}$, $\{T_{u,tgs}, K_{u,tgs}\}K_{u,ws}$, $\{COOKIE\}K_{u,ws}$

Ticket	$T_{p,s} = \{p, s, addr_{ws}, time, life, K_{p,s}\}K_s$
Authenticator	$A_{p,s} = \{p, addr_{ws}, time\}K_{p,s}$
COOKIE	A random value created by the host to protect against replays.

Figure 2-2: Interrealm Extensions to my proposed protocol.

One drawback of this proposed protocol is that for inter-realm authentication the requests are different. This means that if the realm of the user's principal and the realm of the login server are not the same, the protocol has to request a different initial ticket set for the user to decrypt. Therefore, the login server needs to know whether or not to obtain the rcmd service ticket or the shared key service ticket, and the client needs to know to make the request for the rcmd ticket if it is an inter-realm exchange.

In addition, inter-realm authentication means that we have a problem with any offline decryption program, because there is an extra round trip for the cryptographic data required. However, if a terminal emulator, like kermit, has had Charon integrated into it, then this problem goes away, since the application can deal with the whole protocol, including the inter-realm exchange. This protocol is depicted in figure 2-2.

2.4 The Wire Protocol

Although I took some of the ideas for my protocol from rkinit [1], the Charon protocol has been enhanced to meet the requirements of the assumptions placed upon it. Like its predecessor, data is sent in packetized units. A packet consists of a "type byte", a length, and the data. Theoretically, there is no limit on the size of a packet, but to be practical, it is limited to the size of an integer, which is assumed to be 32 bits. Also, it helps to remember that the transfer rate will move 120 bytes of data in one second at 1200 baud. See section 2.5 and table 2.1 for more detailed information on packets.

All data should be in the form of octet strings. Non-string data should be converted to network byte order, which means the most significant byte is first, before being sent over the wire. This is done so that different architectures can share data even if they represent numbers differently.

In addition, before the raw data is transferred, it should be converted to the IA5 alphabet in the encoding scheme described in RFC 1421 [3]. This encoding takes a block of 1-3 bytes of data, and converts it, always, to 4 bytes of encoded data. See appendix A.3 for more information on the IA5 encoding scheme.

The packet is kept separated in its three components, and each component is sent on a line by itself. A line always ends with a carriage-return, newline combination.

The first part of the packet is the "type byte," which is really only six bits long. It is encoded into a single ASCII (IA5) byte, rather than all four, and that is sent alone across the wire.

Second, the length is encoded. The length is that of the octet-string, not the IA5 encoded string. The least significant 1, 2, or 4 bytes[4] of the length is encoded and sent. The smallest number of bytes, between 1, 2, and 4, necessary to send the value should be used. Also, since an IA5 encoding will always produce 4 byte blocks, the encoded length will always have either four or eight bytes of data.[5]

Lastly, the data is converted to IA5, and broken into lines of 64 bytes each. This data, like the length, will always have a length of some multiple of four. It is broken into separate lines because some applications may have limited line lengths. The number 64 was chosen somewhat arbitrarily, however it is a multiple of four and a power of two.[6] Actually, I chose it so that I could easily use X's cut-and-paste feature to test my code.

2.5 Packets

A packet is a unit of data that is sent across the wire from one side to the other. As mentioned, a packet consists of a type byte, length, and data. The type byte is used to specify what the packet contains. This allows the protocol to be event oriented in that it can expect one of a number of different packet types at a particular point in time, and execute different functions given the type of packet received. A packet is defined in table 2.1.

Each packet type has a special usage, all of which is enumerated below. Currently, less than half of the available packet types are in use, but this may change as the protocol develops in the future.

[4] since an integer is assumed to be 32 bits, then "4" bytes refers to the whole value

[5] I could have limited the lengths to 24 bits, which correspond to a single four byte IA5 block, however I decided that flexibility for future protocol additions was more important.

[6] Also, 64 is less than 80, which is the width of standard xterms, ttys, and the VM virtual punch card.

Packet	:= Type Length Data
Type	:= byte (llttttuu)
Length	:= byte[0] ... byte[Llength - 1]
Data	:= byte[0] ... byte[Length-1]
byte	:= msb ... lsb
Llength(ll)	:=
ll = 00	:= Llength = 1
ll = 01	:= Llength = 2
ll = 10	:= Llength = 4
ll = 11	:= Undefined
Types (tttt)	:=
	0x0 Status
	0x1 Client Version
	0x2 Server Version
	0x3 Information Packet
	0x4 Server Kerberos Data
	0x5 Client Kerberos Data
	0x6 Client Authentication Data

Table 2.1: Packet Definitions

`0x0` **Status** Used to relay status of the last command from one side to the other. These can be sent at any time.

`0x1` **Client Version** The packet contains the client version number. When the server gets this, it will choose the lower of this value and its own version number to elect the protocol version number to use.

`0x2` **Server Version** The packet contains the server version number. When the client gets this, it will choose the lower of this number and its own version number to elect the protocol version to use.

`0x3` **Information Structure** This packet contains information from the server. This is described in section 2.6.

`0x4` **Server Kerberos Data** This is used when the login server sends Kerberos data to the client.

`0x5` **Client Kerberos Data** This is for the client to send Kerberos data to the login server, for which the server should forward to the KDC directly.

`0x6` **Client Authentication Data** This is for the client to send its authenticator, TGT, and encrypted cookie to the login server.

2.6 Information Packet Transfer

Since the method of transfer has been determined, the next step was to decide what kind of information will be transfered between the client and server.

The first set of data that needs to be exchanged is version information. In particular, the client needs to know what version of the protocol the server is running, and the server needs to know what version the client is running. If a bad version number is given, then the protocol should signal an error and break contact.

In addition to the cryptographic transfers, which were discussed in section 2.3, the client requires an information packet, which consists of the principal, instance, and realm of the user, the host's "instance" and realm, the server's IP address (which is just in the form of a 32-bit number, so no knowledge of IP is really required), an encryption byte, and the cookie. In future versions of the protocol, such as when Kerberos version 5 is widely deployed, the address could be a krb5 address string, instead of an IP address.

The current version of the protocol (version 2) transfers this data in a single packet. The packet data has the form depicted in table 2.2. This packet can be parsed and unparsed because the data at the beginning is of a constant length, and the data at the end are NULL-terminated strings. This packet is sent from the login server to the terminal.

Bits	*Value*
`0...3`	`cookie`
`4...8`	`IP address`
`9`	`encryption bytes`
`10...`	`principal (nul)`
	`instance (nul)`
	`realm (nul)`
	`host (nul)`
	`host realm (nul)`

Table 2.2: Version 2 information packet structure

Chapter 3

Charon Implementation

The majority of the Charon system is inside a single library. This library has two application entry points, and a number of call-back routines for a user to specify the input/output routines for the library to use.

The library entry points and application calling conventions will be discussed in more detail in the next chapter. Here I will only discuss the library design and implementation – details which are totally internal to the library itself.

3.1 Packet Ordering

The packet types, mentioned in section 2.5, are sent in a particular order to comprise the Charon protocol. A status packet can be sent at any time, and a status of anything other than zero will end the protocol. This allows for either side to break contact at any point if an error occurs.

The version 2 Charon protocol is specified in table 3.1. Each line in the definition is a block of data that needs to be sent. Each block of data may be composed of multiple packets. For example, the block entitled "server Kerberos data" actually consist of two packets, one for the TGT, and one for the service ticket. More on this later.

Table 3.1 includes both the inter-realm protocol and the local-realm protocol. The only difference between them is the data exchange in the middle, where the client sends Kerberos data to the login server and the server replies with the "rcmd" service key. This key is that used by Kerberos for remote commands.

3.2 Control Loop

In order to realize the packet protocol, I developed a dispatch table that allows a compile-time table to be generated. The dispatch loop is event driven. When it enters a state, it sees if there is a

Data From Client		Data From Server
Client Version	$\Rightarrow$	
	$\Leftarrow$	Server Version
	$\Leftarrow$	Information Packet
	$\Leftarrow$	Server Kerberos Data
Client Kerberos Data	$\Rightarrow$	
	$\Leftarrow$	Server Kerberos Data
Client Authentication Data	$\Rightarrow$	
	$\Leftarrow$	Status

Table 3.1: Order of data transfer between client and server machines.

precondition on the state, and if so it runs the specified function. Then it waits for a packet to come in.

When the packet arrives, the loop will dispatch to the appropriate routine for the packet type that arrived, and then update the state. This continues until the protocol has finished or an error occurs (an error can either be locally, or an error status from the other side).

3.3 The Dispatch Table

The dispatch table was designed in order to facilitate the development of the protocol, and to realize the extensibility design goal. By making the protocol event driven, it would be simple to update the protocol at a future date, while keeping backwards compatibility, without having to modify much of the existing code.

A table is a multi-dimensional array, with indices for the mode (client or server), version number, and state. The protocol assumes that the first state (state zero) for any version is the client/server version exchange. Other than that, different versions of the protocol can differ in any way.

The design goal for this section was to make it simple to create newer protocols when appropriate, and make it simple for a programmer to update the code. In addition, I wanted to make it simple for the code to choose which version of the protocol to execute. I decided that a dispatch table would be the most robust approach to meet these goals.

3.4 Modules

The library is broken up into a number of different modules. Each module is supposed to handle a set of functions in a single area. The different modules, each of which will be described later, are the *rpc* module, the *parser* module, the *ia5* module, the *charon* module, an *error* module, the *Kerberos* module, and a *Kerberos library subset* module.

Each module is as self contained as possible, although there might be some inter-dependencies.

By making self-contained modules, it made it easier to test. For example, I could test the IA5 code without needing any other part of the code at all.

A module dependency diagram in shown in figure 3-1.

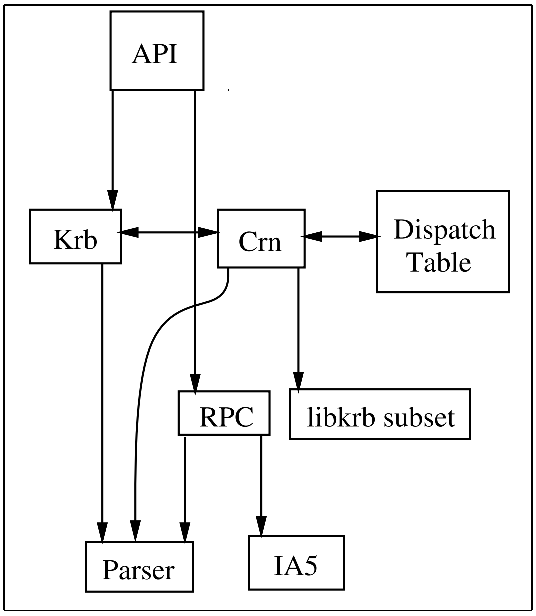

Figure 3-1: Module Dependency Diagram

3.4.1 RPC module

The RPC module is in charge of sending and receiving data. It has a number of functions which an application can use to set some callback procedures, which are used for the actual IO. (These functions are defined in more detail in section 4.1.) If none of the callbacks are set, then the RPC module will use the ANSI standard IO functions.

This module deals with data as packets. A calling function will ask to send a packet, and give the length of the packet and the packet type. This module will send the packet type, then the length, and then the data, as discussed in section 2.4. It is this module that implements that separation.

All data is assumed to be binary. Before it is sent, it is converted to the IA5 alphabet, broken down into lines, and then sent. This work is done mostly in one function:

```
int crn_send_data(u_char *data, u_int datalen, u_char datatype)
```

In addition to sending data, the RPC module also reads data. There are two ways that it can read data: the first is to place it into a pre-existing buffer, and the second is to create a new buffer for the data. To see this through, there are two functions called by other modules. The first one will wait for data to come in and will create a buffer for the data:

```
int crn_wait_for_data(u_char **data, u_int *datalen, u_char *datatype)
```

and the second expects to be able to put the data into an existing buffer:

```
int crn_read_data(u_char *data, u_int *datalen, u_char *datatype)
```

Both of these functions require that the input function block for data, or at least only return with valid data.

The RPC module also has a number of special functions which are used to send special data. Basically, a caller passes in a data type, and this process will call the *rpc* functions to send or read that particular data type. Examples of this type of function include:

```
int crn_rpc_getdat(MSG_DAT *data)
int crn_rpc_senddat(MSG_DAT *data)
int crn_get_status()
```

3.4.2 Parser module

It is the job of the parser module to convert between internal structures and octet strings, for use in sending using the RPC library. For example, the parser contains procedures to convert between integers and octets, which currently assume 32-bit integers.

In addition, the parser module contains the code to create the information packet that gets sent from the login server to the client terminal:

```
int info2octet(charon_t *charon, u_char **data, u_int *datalen)
```

This function takes the internal structure and, depending on the protocol version in use, will create the appropriate data string to send to the client.

The client has an equivalent function, which does the exact reverse operation:

```
int octet2info(u_char *data, charon_t *charon)
```

The calling function must supply the buffer space for the encoded or decoded data, and the encoded or decoded data length should hold the length of that buffer before it is passed into the functions.

3.4.3 IA5 module

IA5 is the International Alphabet Five, a set of 64 ASCII characters that are used to represent six-bit values. The design specified a six-bit clear network, and IA5 was chosen as the representation of all over-wire traffic. These functions convert buffers only, and do not do any line breaking. The *rpc* module is in charge of breaking the encoded string into lines and reconnecting them upon reception.

The IA5 module implements two functions which are used to convert binary data to an IA5 encoded string, and vice-versa:

```
int ia5_encode_data(u_char *data, u_int datalen, char *encoded, u_int *encodedlen)
int ia5_decode_data(char *data, u_int datalen, u_char *decoded, u_int *decodedlen)
```

3.4.4 Error module

This simple module will convert an error number to a printable string that describes what the error is. The returned value is static storage inside the module. The function is called:

`char *crn_err_msg(int error)`

3.4.5 Kerberos Library Subset

The Kerberos library was developed with certain goals in mind, and the implementation has its own breakdown of functionality. However, in order to obtain some of the functionality of Charon, it was necessary to break two particular abstractions in the Kerberos library. These two broken functions are in this module.

The first of these two functions was created to store a ticket in the ticket cache:

`int crn_store_ticket(charon_t *charon, int ticket, int init_cache)`

It takes one argument, `ticket`, to figure out which ticket in the `charon` structure to store and uses the other argument, `init_cache`, to know whether or not it needs to initialize the ticket cache file. This was needed for the client, since the server doesn't send a full KDC response. Rather, the client only gets cyphertext of the ticket. This, however, does contain enough information as far as Kerberos is concerned.

The second function was written because we can't use the Kerberos function `send_to_kdc`. However, in order to obtain the appropriate ticket in the case of an inter-realm authentication, it is necessary for the client to create the packet to send the to remote KDC (see figure 2-2). Therefore, this function was created to obtain the rcmd ticket:

`int crn_get_rcmd_tkt(charon_t *charon)`

3.4.6 Charon module

This module contains all the code to handle the Charon protocol. There is one function, the Charon main loop, which will call these functions based upon the packet type received. The protocol is started, for all intents and purposes, by calling the function

`int crn_main_loop(charon_t *charon)`

This is the main loop, and will send and receive packets as per the version of the protocol in use. It will then call other procedures to process the different types of packets. `ProcessVersion`, `ProcessInfo`, `ProcessClientKDC`, and `ProcessCliAuth` are examples.

3.4.7 Kerberos module

This module contains procedures for Kerberos protocol exchanges, terminal I/O, password fetching, and code to request the initial kerberos tickets. Unlike the Kerberos library subset, this code deals specifically with calls into the Kerberos library that Charon requires, rather than providing the procedural functionality that Kerberos lacks.

The login server needs to send the client terminal two tickets to decrypt: the TGT and some service ticket (either rcmd or the shared key, depending on local- or inter-realm authentication). It does this by using the Kerberos library callback `krb_get_in_tkt`. In fact, the login server uses this twice (once per ticket).

Once the two tickets are obtained, it will enter the Charon main loop and start the communication.

Chapter 4

Application Interface and Examples

This chapter discusses the Applications Programmer Interface (API) of the Charon Library, in depth. It provides examples of applications that use the Charon library, and then it describes modifications to existing programs to use the authentication protocol of Charon.

4.1 API

The API for the Charon library contains a number of functions to set the callbacks for IO functions and to actually start the Charon protocol.

4.1.1 Input/Output

The first set of functions can be used to set the input and output file streams if none of the I/O functions are set:

```
void crn_set_input_file(FILE *file)

    Requires:  file be a valid file, open for reading
    Modifies:  InputFile
    Effects:  Sets the input file to ``file''.
```

```
void crn_set_output_file(FILE *file)

    Requires:  file be a valid file, open for writing
    Modifies:  OutputFile
    Effects:  Sets the output file to ``file''.
```

This next set of functions are used for an application to set callback functions for input and output. There are various modes of input and output, and I tried to create callbacks for as many different modes as I thought applicable. The different modes are I/O by character and I/O by line:

```
void crn_set_putc_fcn(int (*fcn)())

    Requires:  fcn be a function that puts a single character to a
    file, of the form:
    int fcn(char c)

        Effects:  Will output character ‘c’ to the appropriate
        device.

    Modifies:  Putc
    Effects:  Sets the Putc function.

void crn_set_getc_fcn(int (*fcn)())

    Requires:  fcn be a function that gets a single character from a
    file, of the form:
    int fcn()

        Requires:  EOF condition is ignored
        Effects:  Will block until a character is available and
        will return a single character.

    Modifies:  Getc
    Effects:  Sets the Getc function.

void crn_set_puts_fcn(int (*fcn)())

    Requires:  fcn be of form:
    int fcn(char *s, int len)

        Requires:  ‘s’ contain at least ‘len’ characters.
        Effects:  Will output the line held by ‘s’ of length
        ‘len’ to the appropriate output device.  Will always output
        ‘len’ bytes.

    Modifies:  Puts
    Effects:  Sets the Puts function.  This is for kermit compatibility.

void crn_set_fputs_fcn(int (*fcn)())

    Requires:  fcn be a function that puts a line of n chars to the
    file, and includes a CR/LF at the end.  fcn is of the form:
    char *fcn(char *line, FILE *stream)

        Requires:  stream be a file opened for writing.
        Effects:  Output ‘line’ to the file ‘stream’.
```

```
    Modifies: FPuts
    Effects: Sets the FPuts function.
```

```
void crn_set_fgets_fcn(int (*fcn)())
```

```
    Requires: fcn be a function that gets a line from a file, ignoring
    training CR/LF's, of the form:
    char *fcn(char *data, int space, FILE *stream)

        Requires: stream be a file opened for reading and not have
        an EOF condition.
        Effects: Get a line of 'space-1' characters or until a
        newline is reached, and store the data in 'data'.

    Modifies: FGets
    Effects: Sets the FGets function.
```

In addition to just setting the files and callback routines, it is also necessary for the login server application to turn off echoing before starting the server get_ticket function. Therefore, two functions were created to turn off echoing, and to revert the terminal to the state before echoing was turned off:

```
void *crn_kill_echo(int fd)

    Effects: turns off echoing on the file descriptor fd. Returns a
    state structure that should be used to revert the echo.

void crn_revert_echo(void *state)

    Effects: reverts the tty echo 'state'. This will only work once
    on a given state. Once a state is reverted, it cannot be used again.
```

4.1.2 Obtaining Tickets

There are two functions to obtain tickets. One function is for the server, and the other is the client. Once an application gets all the callbacks set up, it needs only to call the appropriate function to initiate the Charon protocol and obtain tickets.

```
int crn_srv_get_tickets(char *principal, char *instance, char *realm, char *hostrealm,
   int lifetime, char *ticketfile, int version, char *enc_ses, des_cblock *key)

    Requires: Echo-ing not be a problem (meaning it should
    be turned off), and that the process be owned by
    the uid who will own the tickets, and also be able
    to read the srvtab file. This can be done by having
    the process's real id be that of the ticket owner, and
    the effective uid be root. In any event, being able to
```

read the srvtab is more important than owning the tickets.
Modifies: key, enc_ses
Effects: Will get kerberos tickets for principal and instance in the specified realm, and put them in the specified file. If the user doesn't exist, can't store the file, or this machine doesn't have an rcmd ticket, then fail with an error to the user. Key is a pointer to the user-defined space to hold a session key, and the session key will be stored there if key is not NULL.
enc_ses is a pointer to a character that will be sent to the client, and it will contain the binary AND of the server and client encrypt_session values upon return. A NULL pointer is equivalent to no encryption information, and nothing will be placed here.
The ticket will be stored in the default location.
Returns CRN_OK if everything goes ok, and the ticket is valid, and all check pass.

int crn_cli_get_tickets(int version, int (*key_proc)(), char *enc_ses, des_cblock *key)

Requires: Echo-ing *not* be a problem (or turned off)
Modifies: key, enc_ses
Effects: Will talk to the server and get the TGT data, decrypt it, and then encrypt the ticket in the rcmd of the server and send it back. Will inform the user if any errors occur.
key_proc is a pointer to a user-defined procedure, expecting arguments: (user, instance, realm, arg, key). If it is NULL, then it will use a pre-defined function to get the user's password and create a DES key.
key is a pointer to the user-defined space to hold a des key, and the session key will be stored there if key is not NULL.
enc_ses is a pointer to an encrypt_session value. It will be sent across to the server, and upon exit will contain the binary AND of the server and client bytes. If it is a NULL pointer, no encryption is assumed.
Returns CRN_OK if everything goes ok.

4.1.3 Errors and Debugging

A function is needed to convert an error number to a printable string, so that a user can understand what an error means:

```
char *crn_err_msg(int error)

    Effects:  Returns a string that explains the error we received
```

There is also one function for debugging purposes. This function takes a filename and will open up a file to write error messages:

```
void crn_rpc_set_debug_file(char *file)

    Modifies:  ErrorFile, charon_debug
    Effects:  Tries to open file for write and will set ErrorFile to
    that file if possible.  Also sets charon_debug if successful.
```

4.2 Applications

The rest of this chapter will discuss a number of applications that I have written or modified to use the Charon protocol. The code for these programs are in the appendices along with the Charon library code. Since the majority of all the work for Charon is done in the library itself, you will notice that these programs are fairly simple, or the changes are minimal. The only thing they do is process commands to set up the necessary state to begin the protocol.

4.2.1 clogin

This program is supposed to let a user login using the Charon protocol for password verification. It is supposed to be run as a shell for a special user that has the same privileges as root. Basically, this program is setup to be the shell of a special user, say "krblogin."

When a user logs in as this special user, this program is run. This program will prompt the user for their real username, and then it will enter the Charon protocol and wait for the client to connect.

Since the Charon library only returns without error when the TGT is successfully obtained (which means that the password was correct), we can use that return value to know that the user successfully typed in their password.

If this is the case, then we have two options. The first option is to emulate the login program itself, add the user to the wtmp and utmp files, add them to the password file[1] and start the user's shell.

[1] This assumes that you are using the Hesiod system of Project Athena, where users are not in /etc/passwd by default, but rather they get added to the password file when they log in.

The second option is to execute some login program that will do all that work, such as the equivalent of `login.krb` from the Kerberos distribution. Currently, I have implemented the first of these two options; it is not however a portable solution, and it would probably be a better solution to use the second of these two options.

4.2.2 ckinit

This program will execute a Charon Initialization. It is run on the login server, after a session has already been instantiated, to obtain a new set of Kerberos tickets. The user supplies a set of options to the program by using command-line flags.

If the program is called without any arguments, it will check the local Kerberos realm for the machine's Kerberos realm, it will check the current ticket cache for the principal, instance, and realm of the user, and, if there are not current tickets, it will get the user's name from the password file and use that.

Optionally, the user can use the command-line options to the program. For example, the "-p" option can be used to set the Kerberos principal (of the form principal.instance@realm). This will override anything that might be in the current ticket cache or the password file. In addition, the user can override the Kerberos realm by using the "-k" option. This will override the default realm.

Tickets normally get stored in the file pointed to by the environment variable KRBTKFILE. However, a user can use the "-f" flag to set a different filename in which to store the tickets.

And finally, the "-l" flag is used to set the lifetime of the tickets. The option to this flag should be the length of time, in minutes, for the ticket to last. If this option is not given, then the default lifetime will be used.

4.2.3 ctelnet

The ctelnet program is one of the client programs modified to use the Charon protocol. It was based upon the code from ktelnet, a Kerberized version of telnet. The Charon protocol was added by creating a new command within the telnet program also called *ckinit*. This "subprogram" within telnet can be executed by escaping back to the `telnet>` prompt and typing *ckinit*. This will initiate the client side of the Charon protocol.

What this will do is use the telnet connection to send data as a Charon client, so when the user connects to a server and tries to use the protocol, they can escape back to the local system and use this telnet command to execute the protocol.

4.2.4 ckermit

Like the ctelnet program, kermit was modified to also have the *ckinit* command embedded inside it. It works in the same manner as the telnet client: A user escapes back to the local machine to start the client Charon protocol.

Chapter 5

Testing Strategies

The code was written using a combination of bottom-up and top-down implementation styles. This way it would be possible to test sections of the code as early as possible and fix any bugs that may have existed before the whole library was connected, since more modules make debugging a difficult task.

The majority of the testing and debugging was done inside xsaber, a C interpreter distributed by Centerline Systems. I used this debugger mostly because it helps find type errors in C, and finds stylistic errors which may lead to problems.

5.1 Testing the Modules

One advantage of a bottom-up implementation is that you can start testing modules right away, since everything that that module depends on has already been implemented. As such, the first modules to be tested were the *ia5* module and *parser* module.

Since xsaber is an interpreter, it was possible to create a program scenario on the spot and see how the implementation executes. For example, in order to test the *ia5* module, I created two buffers. One buffer was for the binary data, and the other buffer was for the encoded data. I then created different lengths of random data and encoded the data, and then decoded the data and compared the results.

I tested the *parser* module in a similar manner, creating test cases, encoding and decoding the test data, and comparing the results by hand. Originally I had a bug in parsing and unparsing the info structure, however that was a problem of pointer arithmatic and was quickly fixed.

5.2 Combined Testing

The next step in testing the code was to combine the system together and try to use it. This was done by having two xsabers running and use the cut-and-paste feature of the X window system to move data from one process to the other.

Some of the bugs I found involved breaking the encoded data into lines, lengths not being correct, and the bug in the parser code that was mentioned previously. In addition, I also found that I had to ignore trailing white-space.

At this point, I was successfully able to obtain a Kerberos ticket for myself. The next test I tried was to obtain an inter-realm ticket. The problem here is that the `send_to_kdc` function in the Kerberos library, as of this writing, does not set the size of the returned packet in the `KTEXT` structure. This means that a function that calls `send_to_kdc` has no way of knowing how much data was received from the KDC.

However, inter-realm authentication requires that the login server send a packet to the remote KDC and then return that packet to the client machine. Therefore, if the `send_to_kdc` call returns successfully, I check the length of the packet. If that length is zero, then I automatically set it to `MAX_KTXT_LEN`, which is the maximum length of a KTEXT packet, as defined in the Kerberos header file.

5.3 Integration Testing

The next step was to test *ckinit* and the modified versions of *kermit* and *telnet*. When I started testing this, I found that there was a bug in my line-breaking code. It was just an error in arithmatic, but it was sending the wrong number of characters, so the protocol was failing.

The reason I didn't see this bug before was because now it was using the callback functions, outputting a single character at a time, rather than using `fputs` to put the line, which just went to the NULL character.

I was able to track the problem down by adding debugging lines that wrote all data transfers to a file, so I could peruse the output in more detail than on a terminal. Even if this problem did exist when I was testing in xsaber, I wouldn't have had a problem because a NULL is a non-printable character, so cut-and-paste wouldn't have grabbed it.

Once that bug was fixed, *ckermit* and *ctelnet* worked properly, and would obtain tickets.

Chapter 6

Conclusions

The conception of the Charon protocol was fairly simple, yet any computer security development inherently contains pitfalls that must be avoided. Because of some assumptions made in the design of the Charon system, extra care was needed to keep the protocol secure. Therefore, it was neccessary to try to think of any possible attack on the system while it was being designed.

However, the ideas that surfaced in the conception of Charon have been floating around for some time. While the difficulty of the concept was fairly minimal, doing it right takes a lot more talent.

The design of the dispatch table took a lot of thought, and actually went through a number of stages of development. It probably could have been done a lot cleaner than my final version, however I thought it was clean enough to be understandable and easily maintained and modified. A lot of the thought was how to actually define the table in an easy manner.

I have come to the conclusion that most communication programs don't give the programmer direct access to character input/output routines. Although this is a minor detail, it was annoying spending hours trying to understand the telnet and kermit code in order to add Charon functionality.

6.1 Future Work

There are a number of possible steps that can be built on top of the Charon system. For one, I don't do any type of encryption of the link once the authentication is made. I tried to allow for encrypting the link with the encryption_session byte in the transfer, however nothing is currently done to use that byte.

This means that some application could be written that actually used that byte and the DES key to encrypt the link once the initial TGT is obtained. This would require the programs on both ends to understand the encryption being used.

I also left the way open to optionally encrypt only certain parts of the session. For example, a session could only encrypt the data from the client to login server, but not the other way. This is one possible use of the session byte.

Also, I have not implemented either the offline client application nor a login server shell that sits between `/bin/login` and `login.krb`. This, too, is something that can easily follow from this thesis.

Charon, as implemented here, is based entirely on Kerberos Version 4. However, much of the work here could be used with Kerberos Version 5. That would require creating a new protocol version, changing some of the parsing routines, and modifying some of the internal Charon routines and structures.

6.2 Acknowledgements

I would like to acknowledge the following people for their help, encouragement, or work that went into this Thesis. It couldn't have happened without them.

Estelle Abrams, Dede and Don Atkins, Ruth Atkins, Richard Basch, Matt Braun, John Carr, Calvin Clark, Mark Eichin, Neil Haller, Ping Huang, Jon Rochlis, Jeff Schiller, Bill Sommerfeld, Theodore T'so, Stan Zanarotti.

Appendix A

Sources to the Charon Library

A.1 crn.c

```
/* crn.c -- This file contains the code to do the state machine for
 * the Charon protocol.  All the state code should be done in this
 * file, and it should call other functions depending on the state,
 * incoming packet type, and application type.  I was considering making
 * it fairly table-driven, as best as possible.
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/crn.c,v $
 * $Author: warlord $
 *
 */

#if !defined(lint) && !defined(SABER)
static char rcsid_crn_c[] = "$Id:  crn.c,v 1.7 93/05/05 11:54:32 warlord Exp $";
#endif

#include <com_err.h>
#include <krb_err.h>
#include <charon_prot.h>
#include <charon_err.h>
#include <rpc.h>
#include <parser.h>
#include "table.h"

static char errbuf[BUFSIZ];

/* This loop is the main dispatch loop of the Charon system.  It
 * called by both the client and server side, and should most likely
 * be table driven.
 *
 * Effects:         Uses the mode (CLIENT_MODE or SERVER_MODE) to know what
 *                  to expect from the other side, and what it should do.
 *                  Uses the version passed in to know which protocol to
 *                  try to use.  The first thing ALL protocols do is
 *                  exchange version info, and they will chose the minimum
 *                  of the two possible versions.
 */
int
crn_main_loop(charon_t *charon)
{
  int state = 0;                    /* state of the machine */
  int retval;                       /* return values from functions */
```

```
int state_changed = 1;          /* did we change state this loop? */
int maxstate;                   /* highest state in table */
u_char pkt;
int pkt_type;
u_char *data;
u_int datalen;
v_table version_table = command_table[charon->mode];
d_table dispatch_table;

/* Lets go into the table.  We will keep going while retval == 0
 * and we have a state to go to.
 */

if (version_table == NULL)
  return (CRN_BAD_VERSION);

do {
  /* The dispatch table can change if the version number
   * changes.  It should ONLY change when processing the
   * version number
   */
  if (charon->version > MAX_VERSION || charon->version < MIN_VERSION) {
    retval = CRN_BAD_VERSION;
    break;
  }

  dispatch_table = version_table[charon->version];

  /* Make sure that this is a valid version table! */
  if (dispatch_table == NULL) {
    retval = CRN_BAD_VERSION;
    break;
  }

  /* Just the maxstate for this table. */
  maxstate = dispatch_table[0].maxstate;

  /* See if we changed state and if we have a function to enter
   * the state.  If both are true, then call that function.
   */
  if (dispatch_table[state].entry != NULL && state_changed)
    (dispatch_table[state].entry)(charon);

  /* Wait for input */
  retval = crn_wait_for_data(&data, &datalen, &pkt);
  pkt_type = pkt;
  if (retval != CRN_OK)
    break;

  charon->data = data;
  charon->datalen = datalen;

  /* Make sure we have a function for this packet type, and if
   * so, then call it!  If not, then set retval = Bad Packet and break
   * from this loop.
   */
  if (dispatch_table[state].fcn[pkt_type] != NULL)
    retval = (dispatch_table[state].fcn[pkt_type])(charon);
  else {
    retval = CRN_UNKNOWN_PKT_TYPE;
    break;
  }

  /* Free this up, if there was anything. */
  if (charon->datalen)
    free(charon->data);

  /* Get the new state, if we were ok */
```

```
    if (retval == CRN_OK) {
      retval = dispatch_table[state].next_state[pkt_type];
      state_changed = retval - state;
      state = retval;
      retval = CRN_OK;
    }

  } while (retval == CRN_OK && state >=0 && state <= maxstate);

  return(retval);
}

static int
CheckStatus(charon_t *charon)
{
  return(crn_read_status((char *)charon->data, charon->datalen));
}

/* Table Procedure  Call to send version to other side. */
static int
SendVersionInfo(charon_t *charon)
{
  return(SendVersion(charon->mode, charon->version));
}

/* Effects:        Sends the version across to the other side
 */
static int
SendVersion(int mode, int version)
{
  u_char data[4];
  int retval;

  int2octet(version, data);

  if (mode == CLIENT_MODE)
    retval = crn_send_data(data, 4, PKT_CVERSION);
  else
    retval = crn_send_data(data, 4, PKT_SVERSION);

  return(retval);
}

/* Modifies:       charon
 * Effects:        sets version to the minimum of version, and the
 *                 version of the other side, which I have in "data".
 *                 If its SERVER_MODE, then send my version to the other
 *                 side, since
 *
 *                 A server-side call means that we need to send the version
 *                 across to the other side.
 *
 *                 returns an error if the minimum is less that MIN_VERSION.
 *                 returns CRN_OK on success.
 */
static int
ProcessVersion(charon_t *charon)
{
  int new_version;
  int other_version;

  octet2int(charon->data, &other_version);

  /* The client would have sent it already.  But the server needs
   * to send this information.
   */
  if (charon->mode == SERVER_MODE)
    SendVersion(SERVER_MODE, charon->version);
```

```
  new_version = min(charon->version, other_version);

  if (new_version < MIN_VERSION)
    return(CRN_BAD_VERSION);

  charon->version = new_version;
  return(CRN_OK);
}

/* Modifies:      charon
 * Effects:       this will take the data input, and unparse it into
 *                the charon structure, using the unparse routine.
 *                Different versions may have different structures, so
 *                if the structure changes, it may be necessary.
 *
 *                This should only be called on a client side.
 *
 *                Returns CRN_OK on success, or an error.
 */
static int
ProcessInfo(charon_t *charon)
{
  return(octet2info(charon->data, charon));
}

/* This procedure sends two packets across, the information
 * packet, and then the two KDC packets.  The two KDC packets
 * are the TGT, and either the rcmd or shared key service tickets
 *
 * This should only be called from a Server side!
 *
 * Requires:      the charon structure be complete, and have the
 *                TGT and either rcmd or shared service key in place
 *                in the KTEXT spots.
 * Effects:       Will send the info, tgt, and tkt2 packets across
 *                to the other side.  It will return CRN_OK on success,
 *                or whatever error gets returned from a sub-procedure.
 */
static int
SendInfoAndKDCPkts(charon_t *charon)
{
  int retval;

  /* First, convert my charon structure to something I can send
   * across, and send the info string.
   */

  if ((retval = info2octet(charon, &(charon->data), &(charon->datalen)))
      != CRN_OK) {
    free(charon->data);
    return(retval);
  }

  if ((retval = crn_send_data(charon->data, charon->datalen, PKT_INFO))
      != CRN_OK) {
    free(charon->data);
    return(retval);
  }

  free(charon->data);

  /* Ok, now we get to send the two KTEXT structures. */
  if ((retval = crn_rpc_put_ktext(charon->tgt, charon->mode)) != CRN_OK)
    return(retval);

  if ((retval = crn_rpc_put_ktext(charon->tkt2, charon->mode)) != CRN_OK)
    return(retval);
```

```
  return(CRN_OK);
}

/* Effects:        Takes the incoming data, which should be the TGT,
 *                 converts it to a KTEXT, then gets another packet,
 *                 which should be another KTEXT packet, which is
 *                 the second ticket, either rcmd.hostinst or the
 *                 shared key.  It then will decrypt these with the
 *                 user's password, and then, if need be, request the
 *                 rcmd ticket.  This will then send the auth data
 *                 back to the server.  The auth data is the rcmd ticket,
 *                 the encrypted cookie, and the encrypted TGT, in that order.
 *
 *                 This should only called on the client side.
 *
 *                 Returns CRN_OK on success, or an error.
 */
static int
ProcessServerKDC(charon_t *charon)
{
  int retval;

  if ((retval = DoProcessServerKDC(charon)) != CRN_OK)
    crn_send_error("Could not process server kerberos data");

  return(retval);
}

static int
DoProcessServerKDC(charon_t *charon)
{
  int retval;
  KTEXT_ST auth;
  CREDENTIALS cred;
  des_cblock key;
  des_key_schedule sched;
  MSG_DAT tgt_data;
  MSG_DAT cookie_data;
  u_char cookie[5];
  u_char tgt_enc_data[MAX_KTXT_LEN];
  u_char cookie_enc_data[MAX_KTXT_LEN];

  /* First, convert the first packet to ktext */
  if ((retval = octet2ktext(charon->data, charon->datalen, charon->tgt))
      != CRN_OK)
    return(retval);

  /* Now, get the second ktext packet */
  if ((retval = crn_rpc_get_ktext(charon->tkt2))
      != CRN_OK)
    return(retval);

  /* Try to decrypt the tickets */
  if ((retval = crn_cli_decrypt_tickets(charon)) != CRN_OK)
    return(retval);

  /* Now see if we need to get an rcmd ticket.  This can be done
   * by checking if the princial realm and host realm are the same.
   * if they are, then we can send the authentication data, otherwise
   * we will have to get the rcmd ticket.
   */

  /* Store tkt2 into a file for reference */
  if ((retval = crn_store_ticket(charon, 0, 1)) != CRN_OK)
    return(retval);

  if (strcmp(charon->realm, charon->hostrealm))
```

```
    if ((retval = crn_get_rcmd_tkt(charon)) != CRN_OK)
      return(retval);

  /* If we've gotten here, then we have valid tickets for a TGT and
   * an rcmd ticket.  So, we need to encrypt everything and ship it
   * across!
   */

  if ((retval = krb_mk_req(&auth, RCMD, charon->hostinst, charon->hostrealm, 0))
      != KSUCCESS)
    return(retval);

  /* Re-encrypt server KDC packet in session key
   * Get credentials from ticket file
   */
  if ((retval = krb_get_cred(RCMD, charon->hostinst, charon->hostrealm, &cred))
      != KSUCCESS)
    return(retval);

  /* Exctract the session key and make the schedule */
  COPY(cred.session, key, sizeof(key));
  if ((retval = des_key_sched(key, sched)) != KSUCCESS)
    return(retval);

  /* Save the session key, if we should */
  if (charon->session != NULL)
    COPY(key, (charon->session), sizeof(des_cblock));

  /* Encrypt the TGT in this session key */
  tgt_data.app_data = tgt_enc_data;
  if ((tgt_data.app_length =
       krb_mk_priv(charon->tgt->dat, tgt_data.app_data, charon->tgt->length,
                   sched, key, &charon->saddr, &charon->saddr)) == -1) {
    return(CRN_MK_PRIV);
  }

  /* Encrypt the cookie in this session key */
  cookie_data.app_data = cookie_enc_data;
  COPY(charon->cookie, cookie, 4);
  cookie[4] = charon->cli_encrypt;
  if ((cookie_data.app_length =
       krb_mk_priv(cookie, cookie_data.app_data, 5,
                   sched, key, &charon->saddr, &charon->saddr)) == -1) {
    return(CRN_MK_PRIV);
  }

  if ((retval = crn_rpc_sendauth(&auth)) != CRN_OK)
    return(retval);

  if ((retval = crn_rpc_senddat(&tgt_data)) != CRN_OK)
    return(retval);

  if ((retval = crn_rpc_senddat(&cookie_data)) != CRN_OK)
    return(retval);

  return(CRN_OK);
}

/* Effects:       This is going to be a request for the rcmd ticket.
 *                Just send it off the the KDC, then take the returned
 *                packet, and send that back to the other side.
 *
 *                This should only be called on the server side.
 *
 *                Returns CRN_OK on success, or an error.
 */
static int
ProcessClientKDC(charon_t *charon)
```

```
{
  int retval;

  /* First, convert the first packet to ktext */
  if ((retval = octet2ktext(charon->data, charon->datalen, charon->tkt2))
      != CRN_OK)
    return(retval);

  /* Now send this off to the KDC */
  if ((retval = send_to_kdc(charon->tkt2, charon->rcmd, charon->hostrealm))
      != KSUCCESS) {
    sprintf(errbuf, "%s while trying to send_to_kdc",
            error_message(ERROR_TABLE_BASE_krb+retval));
    crn_send_error(errbuf);
    return(retval);
  }
  crn_send_success();

  /* XXX -- If there is zero length on this data, then set it to
   * MAX_KTXT_LEN, since the kerberos library may be dain-bramaged
   * and not set the length.
   */
  if (!charon->rcmd->length)
    charon->rcmd->length = MAX_KTXT_LEN;

  /* Ok, return this off to the other side. */
  return(crn_rpc_put_ktext(charon->rcmd, charon->mode));
}

/* Effects:      Expects three packets: a ticket, an encrypted cookie,
 *               and an encrypted tgt. It will verify that the cookie
 *               is the same, that the tgt is for the user in question
 *               (match the username?). If this is the case, then save
 *               the tgt off to a file, and return CRN_OK. Return an
 *               error, otherwise.
 *
 *               This is only called on the server side.
 */
static int
ProcessCliAuth(charon_t *charon)
{
  KTEXT_ST auth;
  AUTH_DAT auth_dat;
  MSG_DAT msg_data;
  MSG_DAT tgt_data;
  MSG_DAT cookie_data;
  u_char tgt_enc_data[MAX_KTXT_LEN];
  u_char cookie_enc_data[MAX_KTXT_LEN];
  des_cblock key;
  des_key_schedule sched;
  int status;

  /* Clear all these structures */
  SBCLEAR(auth);
  SBCLEAR(auth_dat);
  SBCLEAR(tgt_data);
  SBCLEAR(cookie_data);
  BCLEAR(tgt_enc_data);
  BCLEAR(cookie_enc_data);

  tgt_data.app_data = tgt_enc_data;
  cookie_data.app_data = cookie_enc_data;

  /* These are sent in a group */
  if ((status = octet2ktext(charon->data, charon->datalen, &auth)) != CRN_OK)
    return(status);
  if ((status = crn_rpc_getdat(&tgt_data)) != CRN_OK)
    return(status);
```

```
  if ((status = crn_rpc_getdat(&cookie_data)) != CRN_OK)
    return(status);

  /* Read in the request */
  if ((status = krb_rd_req(&auth, RCMD, charon->hostinst,
                           0L, &auth_dat, KEYFILE)) != KSUCCESS) {
    sprintf(errbuf, "%s in krb_rd_req",
            error_message(ERROR_TABLE_BASE_krb+status));
    crn_set_errbuf(errbuf);
    return(status);
  }

  /* copy the session key, and create the key schedule */
  COPY(auth_dat.session, key, sizeof(key));
  if ((status = des_key_sched(key, sched)) != 0) {
    sprintf(errbuf, "%s in des_key_sched",
            error_message(ERROR_TABLE_BASE_krb+status));
    crn_set_errbuf(errbuf);
    return(status);
  }

  /* Copy off the session key, if we should */
  if (charon->session != NULL)
    COPY(key, (charon->session), sizeof(des_cblock));

  /* Decrypt the data and store the TGT. */
  if ((status =
     krb_rd_priv((u_char *)tgt_data.app_data, tgt_data.app_length,
                 sched, key, &charon->saddr, &charon->saddr, &msg_data))
     == KSUCCESS) {
    charon->tgt->length = msg_data.app_length;
    COPY(msg_data.app_data, charon->tgt->dat, msg_data.app_length);
    charon->tgt->dat[charon->tgt->length] = 0;
  } else {
    sprintf(errbuf, "%s in krb_rd_priv of TGT",
            error_message(ERROR_TABLE_BASE_krb+status));
    crn_set_errbuf(errbuf);
    return(status);
  }

  SBCLEAR(msg_data);
  /* Decrypt the cookie and store the charon structure. */
  if ((status =
     krb_rd_priv((u_char *)cookie_data.app_data, cookie_data.app_length,
                 sched, key, &charon->saddr, &charon->saddr, &msg_data))
     == KSUCCESS) {
    if (msg_data.app_length != 5) {
      crn_set_errbuf("Bad Cookie Length.");
      return(CRN_ERR_COOKIE);
    }

    /* Compare the cookie to make sure they are the same! */
    if (CMP(msg_data.app_data, charon->cookie, 4)) {
      crn_set_errbuf("Cookie Mismatch.");
      return(CRN_ERR_COOKIE);
    }
    /* Set the value of the encryption flag */
    charon->cli_encrypt = msg_data.app_data[4];

  } else {
    sprintf(errbuf, "%s in krb_rd_priv of cookie",
            error_message(ERROR_TABLE_BASE_krb+status));
    crn_set_errbuf(errbuf);
    return(status);
  }

  return(CRN_OK);
}
```

A.2 crn_err.c

```
/* crn_err.c -- this file deals with charon errors.  Bascially, it
 * is just user nicety for returning an error message string from
 * an error value.
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/crn_err.c,v $
 * $Author: warlord $
 *
 */

#if !defined(lint) && !defined(SABER)
static char rcsid_crn_err_c[] = "$Id: crn_err.c,v 1.2 93/03/29 01:00:52 warlord Exp $";
#endif

#include <charon_err.h>

/* Effects:         Returns a string that explains the error we received
 */
char *
crn_err_msg(int error)
{
  char *s;

  switch(error) {

  case CRN_OK:
    s = "No Error";
    break;

  case CRN_UNKNOWN_PKT_TYPE:
    s = "Unknown packet type";
    break;

  case CRN_ERR_DATALEN:
    s = "Bad Data Length";
    break;

  case CRN_BAD_VERSION:
    s = "Bad Version Number";
    break;

  case CRN_ALLOC:
    s = "Error allocating memory";
    break;

  case CRN_BAD_PACKET:
    s = "Bad Packet Type";
    break;

  case CRN_BADPW:
    s = "Bad Password";
    break;

  case CRN_KRB_PROT:
    s = "Kerberos Protocol Error";
    break;

  case CRN_MK_PRIV:
    s = "Error in krb_mk_priv";
    break;

  case CRN_ERR_COOKIE:
    s = "Cookie Mismatch";
    break;
```

```
  case CRN_ERROR:
    s = "Generic Charon Error";
    break;

  default:
    s = "Unknown Error";
    break;
  }

  return(s);
}
```

A.3 ia5.c

```
/* ia5.c -- This code will do IA-5 encoding and decoding of strings
 * and characters.
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/ia5.c,v $
 * $Author: warlord $
 *
 */

#if !defined(lint) && !defined(SABER)
static char rcsid_ia5_c[] = "$Id:  ia5.c,v 1.3 93/04/01 20:07:17 warlord Exp $";
#endif

#include <stdio.h>
#include <sys/types.h>
#include <charon_prot.h>
#include <charon_err.h>

#define NUM_CHARS 256           /* There are 256 characters in the character-set */
#define NUM_IA5_CHARS 64        /* 64 characters in IA5 alphabet */

/* IA5 encoding (from RFC 1113):
**    Value Encoding  Value Encoding  Value Encoding  Value Encoding
**        0 A            17 R            34 i            51 z
**        1 B            18 S            35 j            52 0
**        2 C            19 T            36 k            53 1
**        3 D            20 U            37 l            54 2
**        4 E            21 V            38 m            55 3
**        5 F            22 W            39 n            56 4
**        6 G            23 X            40 o            57 5
**        7 H            24 Y            41 p            58 6
**        8 I            25 Z            42 q            59 7
**        9 J            26 a            43 r            60 8
**       10 K            27 b            44 s            61 9
**       11 L            28 c            45 t            62 +
**       12 M            29 d            46 u            63 /
**       13 N            30 e            47 v
**       14 O            31 f            48 w         (pad) =
**       15 P            32 g            49 x
**       16 Q            33 h            50 y
*/

static char EncodeTable[] = {
  'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
  'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',
  'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd',
  'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
  'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
  'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7',
  '8', '9', '+', '/'
  };

static u_char DecodeTable[NUM_CHARS];
static int TableInited = 0;

#define ENCODE_HEX(a) (EncodeTable[a]) /* Convert 6-bit number to character */
#define DECODE_HEX(a) (DecodeTable[a]) /* Convert character to 6-bit number */
#define PAD_CHAR '='              /* Padding character */

/* Get the six highest bits from the first byte, and shift it to make
 * a six-bit number.  This is from the octet string c at offset o
 */
#define CHAR_ONE(c, o) ((c[o] & 0xFC) >> 2)
```

```
/* Get the low two bits, move them over to the top two bits of a six-bit
 * number, and return that
 */
#define CHAR_TWO_OF_TWO(c, o) ((c[o] & 0x03) << 4)

/* Get the low two bits of the first byte and the high 4 bits of the second
 * byte and make that into a single six-bit number.
 */
#define CHAR_TWO(c, o) (CHAR_TWO_OF_TWO(c, o) | ((c[o+1] & 0xF0) >> 4))

/* Get the low 4 bits of the second byte and make it the top four bits
 * of a six-bit number.
 */
#define CHAR_THREE_OF_THREE(c, o) ((c[o+1] & 0x0F) << 2)

/* Get the 4 low bits of the second byte and the two high bits of the
 * third byte, and combine them into a single six-bit number.
 */
#define CHAR_THREE(c, o) (CHAR_THREE_OF_THREE(c, o) | ((c[o+2] & 0xC0) >> 6))

/* Get the low 6 bits of the third byte */
#define CHAR_FOUR(c, o) (c[o+2] & 0x3F)

/* Get the first byte two six-bit numbers, the first, and the top two
 * bits of the second.
 */
#define BYTE_ONE(c, o) ((DECODE_HEX(c[o]) << 2) | \
                        ((DECODE_HEX(c[o+1]) & 0x30) >> 4))

/* Get the second byte from the second and third six-bit values */
#define BYTE_TWO(c, o) (((DECODE_HEX(c[o+1]) & 0x0F) << 4) | \
                        ((DECODE_HEX(c[o+2]) & 0x3C) >> 2))

/* Get the third byte from the third and fourth six-bit values */
#define BYTE_THREE(c, o) (((DECODE_HEX(c[o+2]) & 0x03) << 6) | \
                          DECODE_HEX(c[o+3]))

void ia5_build_decode_table();

/* Requires:    data be valid 8-bit data of length datalen, encoded have
 *              the length of encodedlen, which is at least
 *              (((datalen+2)/3)*4) bytes long
 * Modifies:    encoded and encodedlen
 * Effects:     This will take data, of length datalen, and convert it
 *              from 8-bit to 6-bit using the IA5 encoding scheme
 *              explained in RFC 1113.  It will place the IA5 encoded
 *              data into encoded, and will set encodedlen to the length
 *              of this string.
 */
int
ia5_encode_data(u_char *data, u_int datalen, char *encoded, u_int *encodedlen)
{
  int count;
  int extra_bytes;            /* 0, 1, or 2 */
  int final_length = 0;       /* For encoded length */

  /* Check to make sure we have enough space, just in case someone doesn't
   * understand what a requires clause is all about. ;-)
   */
  if (encodedlen != NULL)
    if (*encodedlen <= ((datalen + 2)/3)*4)
      return (CRN_ERR_DATALEN);


  /* We need to parse the data in blocks of 3 bytes, which will
   * get converted into blocks of 4 characters.  The output will
   * ALWAYS be in blocks of 4 characters, possibly using the PAD_CHAR
   * as filler, if neccessary, to make the 4 characters.
```

```
 */

/* Iterate, Obliterate, Obfuscate, Alleviate.... */
for (count = 0; count < datalen; count += 3) {

  /* Figure out if we really have 3 data bytes, or is this the
   * end.  If its the end, then do we have to pad the data?
   */
  if (datalen - count >= 3)
    extra_bytes = 0;
  else
    extra_bytes = datalen % 3; /* 0, 1, or 2 */

  /* We are now going to take care of 24 bits worth
   * of data.  If we do not HAVE 24 bits worth of
   * data, then we will finish off with padding characters
   *
   * There will ALWAYS be at least one data byte if we
   * have gotten to this point, so we don't have to
   * check that case.
   */

  /* get the six highest bits of the first byte and shift it
   * two places to the right to get the first character.  This
   * Will always be done no matter what extra_bytes has to say.
   * we DID get here in the first place, no?
   */

  encoded[final_length++] = ENCODE_HEX(CHAR_ONE(data, count));

  /* now lets worry about the second character.  If we
   * only have one byte, then we get two characters and
   * two pads.  If we have two bytes, then we get three
   * characters and one pad.  If we have 0 extra bytes
   * then we have no padding at all.
   */

  switch (extra_bytes) {        /* for second character */
  case 1:                           /* We only had one byte, == 2 chars */
    encoded[final_length++] = ENCODE_HEX(CHAR_TWO_OF_TWO(data, count));
    break;
  case 0:
  case 2:
    encoded[final_length++] = ENCODE_HEX(CHAR_TWO(data, count));
    break;
  }

  /* CHARACTER THREE */

  switch (extra_bytes) {        /* for third character */
  case 2:
    encoded[final_length++] = ENCODE_HEX(CHAR_THREE_OF_THREE(data, count));
    break;
  case 0:
    encoded[final_length++] = ENCODE_HEX(CHAR_THREE(data, count));
    break;
  case 1:
    /* Just a pad character */
    encoded[final_length++] = PAD_CHAR;
    break;
  }

  /* CHARACTER FOUR */

  switch (extra_bytes) {        /* for fourth character */
  case 0:
    encoded[final_length++] = ENCODE_HEX(CHAR_FOUR(data, count));
    break;
```

```
    case 1:
    case 2:
      /* Need a pad character */
      encoded[final_length++] = PAD_CHAR;
    }

  } /* for */

  /* Set the encoded length to the length of the string */
  if (encodedlen != NULL)
    *encodedlen = final_length;

  encoded[final_length] = '\0';        /* NULL terminate our char string! */
  return(CRN_OK);
}

/* Requires:     data be valid IA5 of length datalen, decoded be of at least
 *               length (((datalen+3)/4)*3) bytes long, which is stores
 *               in decodedlen.
 * Modifies:     decoded and decodedlen
 * Effects:      This will data data, and IA5 encoded string, and convert
 *               it to the 8-bit data string and place that into decoded,
 *               and set decodedlen to then length of the octet string.
 */
int
ia5_decode_data(char *data, u_int datalen, u_char *decoded, u_int *decodedlen)
{
  int count;
  int extra_bytes = 0;                 /* 0, 1, or 2 */
  int final_length = 0;
  /* This will tell us how many
   *  chars we will have
   * ((octetlen + 2)/3)*4
   */

  /* Build the table if we need to. */
  if (!TableInited)
    ia5_build_decode_table();

  if (datalen%4 != 0)                  /* Must be a multiple of 4! */
    return (CRN_ERR_DATALEN);

  count = (datalen/4)*3;
  if (data[datalen-1] == PAD_CHAR) {
    count--;
    if (data[datalen-2] == PAD_CHAR)
      count--;
  }

  if (decodedlen != NULL)              /* Must have enough space */
    if (*decodedlen < count)
      return (CRN_ERR_DATALEN);

  /* Walk down the string of characters in blocks of four.  If the
   * first is a NULL, then we've gotten to the end.  Check for pad
   * characters, to make sure this works out.
   */

  /* XXX - If a character used is not a valid character as per RFC 1113,
   * it will be treated as a zero!
   */

  for (count = 0; count < datalen && data[count] != '\0'; count +=4) {

    /* Find out how many pad characters there are in this
     * four-character sequece.  There can be 0, 1, or 2
     */
    if (datalen - count > 4)
```

```
      extra_bytes = 0;
    else if (data[count+3] == PAD_CHAR) {
      extra_bytes = 1;
      if (data[count+2] == PAD_CHAR)
            extra_bytes = 2;
    }

    /* First byte */

    decoded[final_length++] = BYTE_ONE(data, count);

    /* Second byte */

    switch (extra_bytes) {
    case 0:
    case 1:
      decoded[final_length++] = BYTE_TWO(data, count);
    case 2:
      break;
    }

    /* Third byte */

    switch (extra_bytes) {
    case 0:
      decoded[final_length++] = BYTE_THREE(data, count);
    case 1:
    case 2:
      break;
    }

  } /* for */

  /* Set the final length of the octen string */
  if (decodedlen != NULL)
    *decodedlen = final_length;

  return(CRN_OK);
}

/* Requires:     True
 * Modifies:     TableInited and DecodeTable
 * Effects:      Builds the decode table from the encode table
 */
void
ia5_build_decode_table()
{
  int i;

  /* Only init the table once */
  if (TableInited)
    return;

  TableInited = 1;

  /* Zero the table */
  BCLEAR(DecodeTable);

  /* Set the values for all the IA5 characters */
  for (i = 0; i < NUM_IA5_CHARS; i++)
    DecodeTable[EncodeTable[i]] = i;

  return;
}
```

A.4 krb.c

```
/* krb.c -- This file contains the code to deal with the kerberos
 * exchange.  It knows about this protocol, and will Do The Right
 * Thing with inter-realm, etc.  Most to the protocol for Charon
 * is in this file
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/krb.c,v $
 * $Author: warlord $
 *
 */

#if !defined(lint) && !defined(SABER)
static char rcsid_krb_c[] = "$Id: krb.c,v 1.15 93/05/06 21:45:32 warlord Exp $";
#endif

#include <stdio.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <com_err.h>
#include <krb.h>
#include <des.h>

#include <signal.h>
#include <setjmp.h>

#ifdef POSIX
#include <termios.h>
#else
#include <sgtty.h>
#endif

#ifdef SERVER
#include <sys/time.h>
#include <netdb.h>
#endif /* SERVER */

#include <charon_prot.h>
#include <charon.h>              /* This includes charon_err.h */
#include <parser.h>
#include <rpc.h>

/* Place to put error messages */
static char errbuf[BUFSIZ];

static jmp_buf env;
static void sig_restore();
static void push_signals();
static void pop_signals();

/* This is a handle structure for the state of a file descriptor
 * when we need to turn off echoing.  No one else needs to know about
 * this.
 */
struct _crn_tty {
  int           fd;              /* File Descriptor for this TTY */
#ifdef POSIX
  struct termios tty;
#else
  struct sgttyb tty;
#endif
};

/* Effects:        reverts echo to the state on the handle that is passed
 *         into this function, and then destroys that handle.
```

```
 */
void
crn_revert_echo(void *state)
{
  struct _crn_tty *tty = state;

  if (state == NULL)
    return;

  /* reset state */
#ifndef POSIX
  ioctl(tty->fd, TIOCSETP, &(tty->tty));
#else
  (void) tcsetattr(tty->fd, TCSAFLUSH, &(tty->tty));
#endif

  /* destroy the handle */
  free(tty);
}

/* Effects:         turns off echoing on the FD, and returns a handle
 *         to a structure to reset that echoing on that FD.
 */
void *
crn_kill_echo(int fd)
{
#ifdef POSIX
  struct termios crn_ttya;
#else
  struct sgttyb crn_ttya;
#endif
  struct _crn_tty *tty = (struct _crn_tty *)malloc(sizeof(struct _crn_tty));

  /* Allocate a handle, save the state, turn off echo ... */
#ifndef POSIX
  ioctl(fd, TIOCGETP, &crn_ttya);
  ioctl(fd, TIOCGETP, &(tty->tty));
  crn_ttya.sg_flags &= ~ECHO;
  ioctl(fd, TIOCSETP, &crn_ttya);
#else
  (void) tcgetattr(fd, &crn_ttya);
  (void) tcgetattr(fd, &(tty->tty));
  crn_ttya.c_lflag &= ~ECHO;
  (void) tcsetattr(fd, TCSAFLUSH, &crn_ttya);
#endif

  /* ... and return the handle */
  tty->fd = fd;
  return((void *)tty);
}

/* This is the key procedure from rkinit.  It might need to be fiddled
 * with to get it to work.  The user can always supply their own
 * key_proc function, of course.
 *
 * Effects:          Will print out key information, and then get the
 *                   user's password, and convert it to a key, and
 *                   return that to the caller.
 */
static int
crn_key_proc(char *user, char *instance, char *realm, char *arg,
             des_cblock key)
{
    charon_t *charon = (charon_t *)arg;
    char password[BUFSIZ];
    int ok = 0;
    void *tty_state;
```

```
    BCLEAR(password);

    /*
     * If the username does not match the aname in the ticket,
     * we will print that too.   Otherwise, we won't.
     */

    printf("Charon initialization (%s)", charon->hostinst);

    printf("\nPassword for %s%s%s@%s:  ", user,
           (instance[0]) ? "." : "", instance, realm);

    fflush(stdout);

    push_signals();
    if (setjmp(env)) {
        ok = -1;
        goto lose;
    }

    tty_state = crn_kill_echo(0);

    if (read(0, password, sizeof(password)) == -1) {
        perror("read");
        ok = -1;
        goto lose;
    }

    if (password[strlen(password)-1] == '\n')
        password[strlen(password)-1] = 0;

     /* Generate the key from the password and destroy the password */

    des_string_to_key(password, key);

lose:
    BCLEAR(password);

    crn_revert_echo(tty_state);

    pop_signals();
    printf("\n");

    return(ok);
}

#ifdef SERVER
/* Effects:        This expects both a TGT and either rcmd or shared
 *                 key to be present, and it will then procede to
 *                 send them across and start the protocol.  Really!
 */
static int
crn_srv_decrypt_2_tkts(char *user, char *instance, char *realm, char *arg,
                       int (*key_proc)(), KTEXT *cipp)
{
  int retval;
  charon_t *info = (charon_t *)arg;

  /* XXX to make saber happy */
  user++; instance++; realm++;

  info->tkt2 = *cipp;

  if ((retval = crn_main_loop(info)) != CRN_OK) {
/*    fprintf(stderr, "Fell out of MainLoop with an error\n"); */
   longjmp(env, retval);
  }
  return(retval);
```

```
}

/* Effects:        This expects to have the TGT, and it will then go off
 *                 and get the rcmd (or shared key) service ticket.  If
 *                 I do get back to here, then I must have successfully
 *                 received the TGT.
 */
static int
crn_srv_decrypt_tgt(char *user, char *instance, char *realm, char *arg,
                    int (*key_proc)(), KTEXT *cipp)
{
  charon_t *info = (charon_t *)arg;
  int same_realm = !strcmp(info->realm, info->hostrealm);

  /* XXX to make saber happy */
  user++; instance++; realm++;

  info->tgt = *cipp;

  /* If we are the same realm, then get rcmd.hostname.  If not,
   * then get krbtgt.realm-of-host.
   */
  krb_get_in_tkt(info->principal, info->instance, info->realm,
                 (same_realm) ? RCMD : KRBTGT,
                 (same_realm) ? info->hostinst : info->hostrealm,
                 1, NULL, crn_srv_decrypt_2_tkts, arg);

  /* If I've returned from here, then things are good. */
  return(CRN_OK);
}
#endif /* SERVER */

/* This is the server-side main call into the charon library.
 *
 * Requires:       Echo-ing not be a problem (meaning it should
 *                 be turned off), and that the process be owned by
 *                 the uid who will own the tickets, and also be able
 *                 to read the srvtab file.  This can be done by having
 *                 the process's real id be that of the ticket owner, and
 *                 the effective uid be root.  In any event, being able to
 *                 read the srvtab is more important than owning the tickets.
 *
 * Modifies:       key, enc_ses
 *
 * Effects:        Will get kerberos tickets for principal and instance
 *                 in the specified realm, and put them in the specified
 *                 file.  If the user doesn't exist, cant store the file,
 *                 or this machine doesnt have an rcmd ticket, then
 *                 fail with an error to the user.  Key is a pointer
 *                 to the user-defined space to hold a session key, and the
 *                 session key will be stored there if it is not NULL.
 *                 enc_ses is a pointer to a character that will be sent
 *                 to the client, and it will contain the binary AND of
 *                 the server and client encrypt_session values upon
 *                 return.  A NULL pointer is equivalent to no encryption
 *                 information, and nothing will be placed here.
 *                 The ticket will be stored in the default location.
 *                 Returns CRN_OK if everything goes ok, and the ticket is
 *                 valid, and all check pass.
 */
int
crn_srv_get_tickets(char *principal, char *instance, char *realm,
                    char *hostrealm, int lifetime, char *ticketfile,
                    int version, char *enc_ses, des_cblock *key)
{
#ifdef SERVER
  /* All this code is only valid if this library has the server code
   * built into it.  Make sure, however, that the call is available
```

```
 * to the user.
 */
charon_t info;
time_t cookie;
int retval;
struct hostent *hp;
char hostname[INST_SZ];
KTEXT_ST rcmd;

/* init error table */
initialize_krb_error_table();

SBCLEAR(info);
SBCLEAR(rcmd);

/* Set it, just in case */
if (enc_ses != NULL) {
  info.srv_encrypt = *enc_ses;
} else
  info.srv_encrypt = CRN_ENC_NONE;

info.cli_encrypt = CRN_ENC_NONE;

/* Set the mode right here. */
info.mode = SERVER_MODE;

/* Copy in the principal, and hostname... These MUST be
 * provided. Also set the username to the value provided and
 * copy in the instance, if provided.
 */
if (!principal || !*principal) {
  fprintf(stderr, "Bad principal\n");
  return(CRN_ERROR);
}
if (gethostname(hostname, INST_SZ - 1) != 0) {
  fprintf(stderr, "Cannot get my hostname\n");
  return(CRN_ERROR);
} else {
  /* Resolve this so we can create a "fake" socket address. */
  if ((hp = gethostbyname(hostname)) == NULL) {
    fprintf(stderr, "Cannot resolve hostname: %s\n", hostname);
    return(CRN_ERROR);
  }

  info.saddr.sin_family = hp->h_addrtype;
  COPY(hp->h_addr, &info.saddr.sin_addr, hp->h_length);

  /* Also get the kerberos hostname for this host. */
  strcpy(info.hostinst, krb_get_phost(hostname));
}
strcpy(info.principal, principal);

if (instance)
  strcpy(info.instance, instance);

/* Copy in the realm if its here. Otherwise, default to local realm,
 * no matter how we get it.
 */
if (realm && *realm)
  strcpy(info.realm, realm);
else
  if (krb_get_lrealm(info.realm, 1) != KSUCCESS)
    strcpy(info.realm, KRB_REALM);

/* If the hostrealm isn't provided, then get the host's realm! */
if (hostrealm && *hostrealm)
  strcpy(info.hostrealm, hostrealm);
else
```

```
    strcpy(info.hostrealm, krb_realmofhost(hostname));

  /* Set the ticket file if we should */
  if (ticketfile && *ticketfile)
    krb_set_tkt_string(ticketfile);

  /* Set the cookie.  This should be set as the current time of this
   * machine.
   */
  cookie = time(0);
  int2octet(cookie, info.cookie);

  /* Find out if we can use the version passed in, otherwise ignore it
   * and choose MAX_VERSION
   */
  if (version < MIN_VERSION || version > MAX_VERSION)
    info.version = MAX_VERSION;
  else
    info.version = version;

  /* Set the session key holder. */
  info.session = key;

  /* Give some space */
  info.rcmd = &rcmd;

  if (geteuid() != 0) {
    fprintf(stderr, "Server not running as root.\n");
    return(CRN_ERROR);
  }

  /* Send this out to stdout... */
  fprintf(stderr, "Charon Initialization...  Waiting for Client Connection.\n");

  /*
   * We need a setjmp here because krb_get_in_tkt ignores the
   * return value of decrypt_tkt.  Thus if we want any of its
   * return values to reach the client, we have to jump out of
   * the routine.
   */

  if (setjmp(env) == 0) {
    /* Get the tickets */
    if ((retval = krb_get_in_tkt(info.principal, info.instance, info.realm,
                                 KRBTGT, info.realm, lifetime,
                                 NULL, crn_srv_decrypt_tgt, (char *)&info))
        != KSUCCESS) {
      crn_send_error(error_message(retval));
    } else
      crn_send_success();

  } else {
    crn_send_error(errbuf);
    retval = CRN_ERROR;
  }
  (void)setuid(0);                 /* make sure I'm root again, if I was before */

  /* Set this value to the bit-and of the two values */
  if (enc_ses != NULL)
    *enc_ses = (info.srv_encrypt & info.cli_encrypt);

  return(retval);
#else /* !SERVER */

  /* Its not a server.  Always return an error! */
  return(CRN_ERROR);
#endif /* SERVER */
}
```

```
/* This is the client-side mail call into the charon library.
 *
 * Requires:      Echo-ing *not* be a problem (or turned off)
 *
 * Modifies:      key, enc_ses
 *
 * Effects:       Will talk to the server and the the TGT data,
 *                decrypt it, and then encrypt the ticket in the
 *                rcmd of the server and send it back.  Will inform
 *                the user if any errors occur.
 *                key_proc is a pointer to a user-defined procedure, expecting
 *                arguments: (user, instance, realm, arg, key).  If it is
 *                NULL, then use pre-defined function to get
 *                the user's password and create a DES key.
 *                key is a pointer to the user-defined space to hold a
 *                des key, and the session key will be stored there if
 *                it is not NULL.
 *                enc_ses is a pointer to an encrypt_session value.  It
 *                will be sent across to the server, and upon exit will
 *                contain the binary AND of the server and client bytes.
 *                If it is a NULL pointer, no encryption is assumed.
 *                Returns CRN_OK if everything goes ok.
 */
int
crn_cli_get_tickets(int version, int (*key_proc)(), char *enc_ses,
                    des_cblock *key)
{
  KTEXT_ST tgt, tkt2, rcmd;
  int status, status2;
  charon_t info;

  SBCLEAR(info);
  SBCLEAR(tgt);
  SBCLEAR(tkt2);
  SBCLEAR(rcmd);

  /* init error table */
  initialize_krb_error_table();

  /* This is to prevent losing current tickets */
  krb_set_tkt_string("/dev/null");

  /* Set it, just in case */
  if (enc_ses != NULL) {
    info.cli_encrypt = *enc_ses;
  } else
    info.cli_encrypt = CRN_ENC_NONE;

  info.srv_encrypt = CRN_ENC_NONE;

  /* Set the mode.  No, not the mood, the MODE */
  info.mode = CLIENT_MODE;

  /* Set the key proc */
  if (key_proc != NULL)
    info.key_proc = key_proc;
  else
    info.key_proc = crn_key_proc;

  /* Set the version, if we can. */
  if (version < MIN_VERSION || version > MAX_VERSION)
    info.version = MAX_VERSION;
  else
    info.version = version;

  /* Set the session key space */
```

```
  info.session = key;

  /* Put in data for the tickets. */
  info.tgt = &tgt;
  info.tkt2 = &tkt2;
  info.rcmd = &rcmd;

  status = crn_main_loop(&info);
  status2 = crn_get_status();

#ifndef DEBUG
  /* Destroy the local tickets we got -- they're not useful anyways */
  if (info.obtain_ticket)
    dest_tkt();
#endif

  /* Set this return value */
  if (enc_ses != NULL)
    *enc_ses = (info.cli_encrypt & info.srv_encrypt);

  if (status == CRN_OK)
    return(status2);

  return(status);
}

int
crn_cli_decrypt_tickets(charon_t *charon)
{
  KTEXT cip = charon->tgt;
  C_Block key;            /* Key for decrypting cipher */
  Key_schedule key_s;

  /* generate a key */
  {
    register int rc;
    rc = (*charon->key_proc)(charon->principal, charon->instance,
                             charon->realm, (char *)charon, key);
    if (rc)
      return(rc);
  }

  des_key_sched(key, key_s);

  /* Decrypt information from KDC */
  des_pcbc_encrypt((C_Block *)cip->dat,(C_Block *)cip->dat,
                   (long) cip->length, key_s, key, 0);

  /* Decrypt the second ticket */
  cip = charon->tkt2;
  des_pcbc_encrypt((C_Block *)cip->dat,(C_Block *)cip->dat,
                   (long) cip->length, key_s, key, 0);

  /* Get rid of all traces of key */
  BCLEAR(key);
  BCLEAR(key_s);

  return(CRN_OK);
}

/* Effects:     Takes the character c and will encrypt it in the
 *              session key supplied and return that value.  It will
 *              Take the value of mode and enc_ses to figure out
 *              how to encrypt this character.
 */
char
crn_encrypt_char(char c, des_cblock session, int mode, char enc_ses)
{
```

```
  if ((mode == CLIENT_MODE && enc_ses == CRN_ENC_CLI) ||
      (mode == SERVER_MODE && enc_ses == CRN_ENC_SRV) ||
      enc_ses == CRN_ENC_BOTH)
    return(c + 1);
  else
    return(c);
}

/* Effects:         Takes the character c and will decrypt it using the
 *                  session key provided, and return that value.
 */
char
crn_decrypt_char(char c, des_cblock session, int mode, char enc_ses)
{
  if ((mode == CLIENT_MODE && enc_ses == CRN_ENC_SRV) ||
      (mode == SERVER_MODE && enc_ses == CRN_ENC_CLI) ||
      enc_ses == CRN_ENC_BOTH)
    return(c - 1);
  else
    return(c);
}

/* Effects:         sets the errbuf to msg */
int
crn_set_errbuf(char *msg)
{
  strncpy(errbuf, msg, sizeof(errbuf));
}

#ifdef POSIX
static void (*old_sigfunc[NSIG])();
#else
static int (*old_sigfunc[NSIG])();
#endif POSIX

static void push_signals()
{
    register i;
    for (i = 0; i < NSIG; i++)
        old_sigfunc[i] = signal(i,sig_restore);
}

static void pop_signals()
{
    register i;
    for (i = 0; i < NSIG; i++)
        signal(i,old_sigfunc[i]);
}

static void sig_restore(sig,code,scp)
    int sig,code;
    struct sigcontext *scp;
{
    longjmp(env,1);
}
```

A.5 lib_krb.c

```c
/* lib_krb.c -- This file contains functions that should be in
 * libkrb.a, * or whatever kerberos library is being used, but which
 * needs to be called * internally, since I don't really get the
 * functionality of the kerberos * library that I wan't to have....
 * Therefore I have created these functions in order to get the
 * functionality I needed.  A lot of it is taken out of the kerberos
 * library.
 *
 * Created by: Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/lib_krb.c,v $
 * $Author: warlord $
 *
 */

#if !defined(lint) && !defined(SABER)
static char rcsid_lib_krb_c[] = "$Id: lib_krb.c,v 1.4 93/04/01 19:59:07 warlord Exp $";
#endif

#include <sys/time.h>
#include <krb.h>
#include <des.h>
#include <charon_prot.h>
#include <charon_err.h>
#include <krb_prot.h>
#include <stdio.h>

extern int krb_debug;

/* Requires:    ticket cache to have been initialized already.
 * Effects:     takes a ticket an stores it in the ticket cache file.
 *              If init_cache == 1, then initialize the ticket cache.
 *              If ticket == 0, then use charon->tkt2, else use
 *              ticket->rcmd.
 *
 *              This was taken from krb_get_in_tkt.c
 */
int
crn_store_ticket(charon_t *charon, int ticket, int init_cache)
{
  char *ptr;
  KTEXT cip;
  C_Block ses;                  /* Session key for tkt */
  char s_name[SNAME_SZ];
  char s_instance[INST_SZ];
  char rlm[REALM_SZ];
  int lifetime;
  int kvno;                     /* Kvno for session key */
  KTEXT_ST tkt_st;
  KTEXT tkt = &tkt_st;          /* Current ticket */
  struct timeval t_local;
  int kerror;

  if (!ticket)
    cip = charon->tkt2;
  else
    cip = charon->rcmd;

  ptr = (char *) cip->dat;

  /* extract session key */
  COPY(ptr,ses,8);
  ptr += 8;

  if ((strlen(ptr) + (ptr - (char *) cip->dat)) > cip->length)
```

```
    return(CRN_BADPW);

  /* extract server's name */
  (void) strncpy(s_name,ptr, sizeof(s_name)-1);
  s_name[sizeof(s_name)-1] = '\0';
  ptr += strlen(s_name) + 1;

  if ((strlen(ptr) + (ptr - (char *) cip->dat)) > cip->length)
    return(CRN_BADPW);

  /* extract server's instance */
  (void) strncpy(s_instance,ptr, sizeof(s_instance)-1);
  s_instance[sizeof(s_instance)-1] = '\0';
  ptr += strlen(s_instance) + 1;

  if ((strlen(ptr) + (ptr - (char *) cip->dat)) > cip->length)
    return(CRN_BADPW);

  /* extract server's realm */
  (void) strncpy(rlm,ptr, sizeof(rlm));
  rlm[sizeof(rlm)-1] = '\0';
  ptr += strlen(rlm) + 1;

  /* extract ticket lifetime, server key version, ticket length */
  /* be sure to avoid sign extension on lifetime! */
  lifetime = (unsigned char) ptr[0];
  kvno = (unsigned char) ptr[1];
  tkt->length = (unsigned char) ptr[2];
  ptr += 3;

  if ((tkt->length < 0) ||
      ((tkt->length + (ptr - (char *) cip->dat)) > cip->length))
    return(CRN_BADPW);

  /* extract ticket itself */
  COPY(ptr,(tkt->dat),tkt->length);
  ptr += tkt->length;

  /* Pass KDC time stamp -- I'm using the Cookie to verify it. */
  ptr += 4;

  /* initialize ticket cache, if we should */
  if (init_cache) {
    char ticketfile[200];
    /* First, set the ticket file name */

    sprintf(ticketfile, "%s%s.crn", TMP_TKT_FILE, charon->principal);
    krb_set_tkt_string(ticketfile);

    if (in_tkt(charon->principal,charon->instance) != KSUCCESS)
      return(INTK_ERR);

    charon->obtain_ticket = 1;
  }

  /* Get local time */
  (void) gettimeofday(&t_local,(struct timezone *) 0);

  /* stash ticket, session key, etc. for future use */
  if ((kerror = save_credentials(s_name, s_instance, rlm, ses,
                                 lifetime, kvno, tkt, t_local.tv_sec))
      != KSUCCESS)
    return(kerror);

  return(CRN_OK);
}

/* This is in the kerberos library */
```

```
extern int swap_bytes;

/* Effects:        This will get an rcmd ticket from the kerberos server
 *         by going across the rpc connection and asking for it.  It will
 *         then store that ticket in a file.
 *         returns CRN_OK on success, or an error.
 */
int
crn_get_rcmd_tkt(charon_t *charon)
{
  KTEXT_ST pkt_st;
  KTEXT pkt = & pkt_st; /* Packet to KDC */
  KTEXT_ST rpkt_st;
  KTEXT rpkt = &rpkt_st;            /* Returned packet */
  KTEXT cip = charon->rcmd;     /* Returned Ciphertext */
  CREDENTIALS cr;
  Key_schedule key_s;
  long time_ws = 0;
  int msg_byte_order;
  int kerror;
  unsigned long rep_err_code;
  static struct timeval tt_local = { 0, 0 };

  BCLEAR(pkt);
  BCLEAR(rpkt);

  /* Reset this variable */
  swap_bytes = 0;

  /* Create skeleton of packet to be sent */
  (void) gettimeofday(&tt_local,(struct timezone *) 0);

  pkt->length = 0;

  /*
   * Look for the session key (and other stuff we don't need)
   * in the ticket file for krbtgt.realm@lrealm where "realm"
   * is the service's realm (passed in "realm" argument) and
   * lrealm is the realm of our initial ticket.  If we don't
   * have this, we will try to get it.
   */

  if ((kerror = krb_get_cred(KRBTGT,charon->hostrealm,charon->realm,&cr))
     != KSUCCESS) {
         return(kerror);
     }

  /*
   * Make up a request packet to the "krbtgt.realm@lrealm".
   * Start by calling krb_mk_req() which puts ticket+authenticator
   * into "pkt".  Then tack other stuff on the end.
   */

  kerror = krb_mk_req(pkt,KRBTGT,charon->hostrealm,charon->realm,0L);

  if (kerror)
    return(AD_NOTGT);

  /* timestamp */
  COPY(&time_ws,(pkt->dat+pkt->length),4);
  pkt->length += 4;
  *(pkt->dat+(pkt->length)++) = (char) 1;
  (void) strcpy((char *) (pkt->dat+pkt->length),RCMD);
  pkt->length += 1 + strlen(RCMD);
  (void) strcpy((char *)(pkt->dat+pkt->length),charon->hostinst);
  pkt->length += 1 + strlen(charon->hostinst);

  rpkt->length = 0;
```

```
  /* Send the request to the local ticket-granting server
   * THIS IS SENT VIA THE RPC PORT
   *
   * Also, I *know* that this is only going to be a client!
   */
  crn_rpc_put_ktext(pkt, CLIENT_MODE);
  if ((kerror = crn_get_status()) != CRN_OK)
    return(kerror);
  crn_rpc_get_ktext(rpkt);

  /* check packet version of the returned packet */
  if (pkt_version(rpkt) != KRB_PROT_VERSION )
    return(CRN_KRB_PROT);

  /* Check byte order */
  msg_byte_order = pkt_msg_type(rpkt) & 1;
  swap_bytes = 0;
  if (msg_byte_order != HOST_BYTE_ORDER)
    swap_bytes++;

  switch (pkt_msg_type(rpkt) & ~1) {
  case AUTH_MSG_KDC_REPLY:
    break;
  case AUTH_MSG_ERR_REPLY:
    COPY(pkt_err_code(rpkt), &rep_err_code, 4);
    if (swap_bytes)
      swap_u_long(rep_err_code);
    return(rep_err_code);

  default:
    return(CRN_KRB_PROT);
  }

  /* Extract the ciphertext */
  cip->length = pkt_clen(rpkt);      /* let clen do the swap */

  COPY(pkt_cipher(rpkt),(cip->dat),cip->length);

#ifndef NOENCRYPTION
  /* Attempt to decrypt it */

  key_sched(cr.session,key_s);
  if (krb_debug)  printf("About to do decryption ...");
  pcbc_encrypt((C_Block *)cip->dat,(C_Block *)cip->dat,
                (long) cip->length,key_s,cr.session,0);
#endif /* !NOENCRYPTION */
  /* Get rid of all traces of key */
  BCLEAR(cr.session);
  BCLEAR(key_s);

  /* Continue to parse this ticekt and then store it */
  return(crn_store_ticket(charon, 1, 0));
}
```

A.6 parser.c

```
/* parser.c -- This will have the parse and unparse routines to convert
 * internal structures into something that can be sent over the
 * wire and retreived.
 *
 * Created by:     Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/parser.c,v $
 * $Author: warlord $
 *
 */

#if !defined(lint) && !defined(SABER)
static char rcsid_parser_c[] = "$Id: parser.c,v 1.4 93/05/05 11:55:10 warlord Exp $";
#endif

#include <stdio.h>
#include <sys/types.h>
#include <krb.h>
#include <charon_prot.h>
#include <charon_err.h>

/* Requires:       octet be of at least size 4 bytes, and int >= 0, and int
 *                 be of at most size 4 bytes
 * Modifies:       octet
 * Effects:        converts the integer num to an MSB-first represented
 *                 octet string, and put this into octet.
 */
void
int2octet(int num, u_char *octet)
{
  int i;

  for (i = 3; i >=0; i--) {
    octet[i] = num%256;
    num /= 256;
  }
}

/* Requires:       octet be of size 4 bytes.
 * Modifies:       num
 * Effects:        converts an MSB-first octet string to an integer.
 */
void
octet2int(u_char *octet, int *num)
{
  int i;

  for (*num = 0, i = 0; i < 4; i++) {
    *num *= 256;
    *num += octet[i];
  }
}

/* This is a description of the format of the octet string from
 * the info structure, for each version.  If there is a new protocol
 * version, then this code might need to be changed.
 *
 * For version 2:
 *
 *  0...3 | 4...8 | 9 | 10 ... end
 * cookie |IP addr| e | principal 0 instance 0 realm 0 host 0 hostrealm 0
 */

/* Modifies:       data, datalen
 * Effects:        will convert the charon structure to an octet string
```

```
 *                  of a form capable of being sent across the wire.
 *                  The format should match that of octet2info.
 *                  Will malloc the space needed.
 *                  Returns CRN_OK on success, error otherwise.
 */
int
info2octet(charon_t *charon, u_char **data, u_int *datalen)
{
  *datalen = 0;                          /* initialize length to zero */

  switch (charon->version) {

  case 2:
    /* allocate max size == 4 + sum(sizes of principal, inst, realm, hostinst,
     * and hostrealm).
     */
    *data = (u_char *)malloc(4 + sizeof(struct sockaddr_in) +
                             sizeof(charon->principal) +
                             sizeof(charon->instance) +
                             sizeof(charon->realm) +
                             sizeof(charon->hostinst) +
                             sizeof(charon->hostrealm));

    if (*data == NULL)
      return(CRN_ALLOC);

    /* Ok, now that its actually allocated, lets convert it all.
     * First, convert the cookie
     */
    COPY(charon->cookie, *data, 4);
    *datalen += 4;

    /* Copy in the address */
    COPY(&(charon->saddr.sin_addr), *data+*datalen,
         sizeof(charon->saddr.sin_addr));
    *datalen += sizeof(charon->saddr.sin_addr);

    /* Put in the srv_encrypt byte */
    *(*data+*datalen) = charon->srv_encrypt;
    *datalen++;

    /* Now, copy each of the other parts.  Make sure to keep the
     * NULL separation.  Start with the principal.
     */
    strcpy(*data+*datalen, charon->principal);
    *datalen += strlen(charon->principal) + 1;

    /* Copy the instance */
    strcpy(*data+*datalen, charon->instance);
    *datalen += strlen(charon->instance) + 1;

    /* And the realm */
    strcpy(*data+*datalen, charon->realm);
    *datalen += strlen(charon->realm) + 1;

    /* Now the host instance */
    strcpy(*data+*datalen, charon->hostinst);
    *datalen += strlen(charon->hostinst) + 1;

    /* And finally the host realm */
    strcpy(*data+*datalen, charon->hostrealm);
    *datalen += strlen(charon->hostrealm) + 1;

    break;

  default:
    return(CRN_BAD_VERSION);
  }
```

```
  return(CRN_OK);
}

/* Requires:     data be in the correct format for the version.
 * Modifies:     charon
 * Effects:      This function will convert an octet string to an internal
 *               structure. It is the reverse of info2octet, and it should
 *               match at all times. The version of the software can be used
 *               as a flag if the structure changes.
 *               Returns CRN_OK on success, error otherwise.
 */
int
octet2info(u_char *data, charon_t *charon)
{
  char *char_p;

  switch(charon->version) {

  case 2:
    /* Pull out the cookie */
    COPY(data, charon->cookie, 4);
    data += 4;

    /* Now get the server address */
    COPY(data, &(charon->saddr.sin_addr), sizeof(charon->saddr.sin_addr));
    data += sizeof(charon->saddr.sin_addr);

    /* Get the server's encryption byte */
    charon->srv_encrypt = *data;
    data++;

    /* Now get the principal */
    char_p = charon->principal;
    do {
      *(char_p++) = *data;
    } while (*(data++) != NULL);

    /* And the instance */
    char_p = charon->instance;
    do {
      *(char_p++) = *data;
    } while (*(data++) != NULL);

    /* And the realm */
    char_p = charon->realm;
    do {
      *(char_p++) = *data;
    } while (*(data++) != NULL);

    /* And the hostname */
    char_p = charon->hostinst;
    do {
      *(char_p++) = *data;
    } while (*(data++) != NULL);

    /* And the hostrealm */
    char_p = charon->hostrealm;
    do {
      *(char_p++) = *data;
    } while (*(data++) != NULL);

    break;

  default:
    return (CRN_BAD_VERSION);
  }
```

```
  return(CRN_OK);
}

/* Effects:        Takes an octet string and puts it into a ktext
 *                 structure.
 */
int
octet2ktext(u_char *octet, int datalen, KTEXT pkt)
{
  COPY(octet, pkt->dat, datalen);
  pkt->length = datalen;
  return(CRN_OK);
}
```

A.7 rpc.c

```
/* Rpc.c -- This file contains the code to do the rpc over the serial
 * line.  It will do actual to the transmissions over the port.  It
 * keeps an internal set of file descriptors for input and output (they
 * do not have to be the same).
 *
 * Created by:     Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/rpc.c,v $
 * $Author: warlord $
 *
 */

#if !defined(lint) && !defined(SABER)
static char rcsid_rpc_c[] = "$Id: rpc.c,v 1.6 93/04/01 19:59:23 warlord Exp $";
#endif

#include <stdio.h>
#include <ia5.h>
#include <parser.h>
#include <charon_prot.h>
#include <charon_err.h>

#if defined(vax) || defined(ibm032)
char *fgets(), *fputs();
#endif

int charon_debug = 0;

/* default to stdin and stdout */
static FILE *InputFile = stdin, *OutputFile = stdout, *ErrorFile = stderr;
static int (*Putc)() = NULL, (*Getc)() = NULL;
static int (*Puts)() = NULL;
static int (*FPuts)() = NULL, (*FGets)() = NULL;

/* Requires:      file be a valid file, open for reading
 * Modifies:      InputFile
 * Effects:       Sets the input file to "file".
 */
void
crn_set_input_file(FILE *file)
{
  InputFile = file;
  return;
}

/* Requires:      file be a valid file, open for writing
 * Modifies:      OutputFile
 * Effects:       Sets the output file to "file".
 */
void
crn_set_output_file(FILE *file)
{
  OutputFile = file;
  return;
}

/* Requires:      fcn be a function that puts a single character to a
 *                file, of the form int fcn(char c)
 * Modifies:      Putc
 * Effects:       Sets the Putc function.
 */
void
crn_set_putc_fcn(int (*fcn)())
{
  Putc = fcn;
```

```
}

/* Requires:     fcn be a function that gets a single character from a
 *               file, of the form int fcn()
 * Modifies:     Getc
 * Effects:      Sets the Getc function.
 */
void
crn_set_getc_fcn(int (*fcn)())
{
  Getc = fcn;
}

/* Requires:     fcn be of form int fcn(char *s, int len)
 * Effects:      Sets the Puts function.  This is for kermit compatibility.
 */
void
crn_set_puts_fcn(int (*fcn)())
{
  Puts = fcn;
}

/* Requires:     fcn be a function that puts a line of n chars to the
 *               file, and includes a CR/LF at the end. fc is of the form
 *               char *fcn(char *line, FILE *stream)
 * Modifies:     FPuts
 * Effects:      Sets the FPuts function.
 */
void
crn_set_fputs_fcn(int (*fcn)())
{
  FPuts = fcn;
}

/* Requires:     fcn be a function that gets a line from a file, ignoring
 *               training CR/LF's, of the form
 *               char *fcn(char *data, int space, FILE *stream)
 * Modifies:     FGets
 * Effects:      Sets the FGets function.
 */
void
crn_set_fgets_fcn(int (*fcn)())
{
  FGets = fcn;
}

/* Effects:      Will take the KTEXT packet and send it across to the
 *               other side.  It will vary the type depending on whether
 *            the srv is SERVER_MODE or CLIENT_MODE.
 *               Retuns CRN_OK on success, otherwise an error.
 */
int
crn_rpc_put_ktext(KTEXT pkt, int srv)
{
  return (crn_send_data(pkt->dat, pkt->length,
                        (srv == CLIENT_MODE) ? PKT_CKDC : PKT_SKDC));
}

/* Modifies:     pkt
 * Effects:      Will get a KTEXT packet from the other side.  It will
 *               set equal to a boolean of whether the types matched or
 *               not.
 *               Retuns CRN_OK on success, otherwise an error.
 */
int
crn_rpc_get_ktext(KTEXT pkt)
{
  u_char type;
```

```
  pkt->length = MAX_KTXT_LEN; /* set to max size for default */

  return(crn_read_data(pkt->dat, (u_int *)&(pkt->length), &type));
}

/* Effects:        Sends an AUTH_DAT packet to the other side.
 */
int
crn_rpc_sendauth(KTEXT auth)
{
  return(crn_send_data(auth->dat, auth->length, PKT_CAUTH));
}

/* Effects:        Send a MSG_DAT packet to the other side.
 */
int
crn_rpc_senddat(MSG_DAT *data)
{
  return(crn_send_data(data->app_data, data->app_length, PKT_CAUTH));
}

/* Requires:       Space to have been provided already.
 * Effects:        Reads in a MSG_DAT structure from the other side and puts
 *                 it in the space provided.
 */
int
crn_rpc_getdat(MSG_DAT *data)
{
  int status;
  u_char type;

  data->app_length = MAX_KTXT_LEN; /* initialize to largest size. */

  if ((status = crn_read_data(data->app_data, &data->app_length, &type))
      != CRN_OK)
    return(status);

  if (type != PKT_CAUTH)
    return(CRN_BAD_PACKET);

  return(CRN_OK);
}

/* Effects:        Reads a packet and expects it to be a status packet, and
 *                 will return CRN_OK on good status, or an error.
 */
int
crn_get_status()
{
  char msg[BUFSIZ];
  u_char type;
  u_int len = sizeof(msg);
  int status;

  if ((status = crn_read_data((u_char *)msg, &len, &type)) != CRN_OK)
    return(status);

  if (type != PKT_STATUS)
    return(CRN_BAD_PACKET);

  return(crn_read_status(msg, len));
}

/* Effects:        Parse a data and length, and return CRN_OK on error, or
 *                 CRN_ERROR and print an error message.
 */
int
```

```
crn_read_status(char *msg, u_int len)
{
  if (len == 0)
    return(CRN_OK);

  fprintf(stderr, "%s\n", msg);
  return(CRN_ERROR);
}

/* Effects:        Send an OK status to the other side.
 */
void
crn_send_success()
{
  crn_send_data(NULL, 0, PKT_STATUS);
}

/* Effects:        Send an error message across to the other side.
 */
void
crn_send_error(char *msg)
{
  crn_send_data((u_char *)msg, strlen(msg)+1, PKT_STATUS);
}

/* Requires:       crn_read_line blocks.
 * Modified:       data, datalen
 * Effects:        This will wait for an incoming packet, somehow, and then
 *                 it will read in the packettype, packet length, and the
 *                 packet itself.  It will allocate enough data for the packet
 *                 and return the packet type to the caller.
 */
int
crn_wait_for_data(u_char **data, u_int *datalen, u_char *datatype)
{
  u_char type_byte;
  int llen;
  char type[5];
  char enc_length[9];
  u_char length[4];
  int retval;
  u_int count, encodedlen;

  /* First, read the packet type byte (should only be one character) */
  if ((retval = crn_read_line(type, sizeof(type))) != 1)
    return(CRN_BAD_PACKET);

  /* This is special, since I really do know the IA5 encoding scheme here */
  type[1] = type[2] = type[3] = '=';
  type[4] = '\0';

  /* Decode the type-byte, and get the llen */
  encodedlen = 5;
  if ((retval = ia5_decode_data(type, 4, &type_byte, &encodedlen)) != CRN_OK)
    return(retval);
  *datatype = PKT_TYPE(type_byte);
  llen = PKT_LLENGTH(type_byte);

  /* Read in the length field */
  count = crn_read_line(enc_length, sizeof(enc_length));

  /* And convert it back ... */
  BCLEAR(length);
  encodedlen = 4;
  if ((retval = ia5_decode_data(enc_length, count, length + (4 - llen),
                                &encodedlen)) != CRN_OK) {
    return(retval);
  }
```

```
  /* ... to an integer */
  octet2int(length, (int *)datalen);

  if (charon_debug)
    fprintf(ErrorFile, "crn_len:  %d\n", *datalen);

  /* Now allocate and clear the space for this */
  *data = (u_char *)malloc(*datalen + 1);
  ZERO(*data, *datalen);

  return(crn_read_packet_data(*data, *datalen));
}

/* Requires:      Valid Input and Output Files, valid 8-bit data and datalength,
 *                and a valid type
 * Modifies:      Output stream
 * Effects:       Will convert the data to a 6-bit, IA5 encoding, and send
 *                that to the output file.
 *                Returns 0 on success, or an error code, otherwise
 */
int
crn_send_data(u_char *data, u_int datalen, u_char datatype)
{
  u_char type_byte;
  int llen, retval;
  char type[5];
  char enc_length[9];
  u_char length[4];
  char *coded_data;
  u_int count, encodedlen;

  /* Encoded the length into an octet */
  int2octet(datalen, length);

  llen = 4;
  if (datalen < 0x010000)
    llen = 2;
  if (datalen < 0x100)
    llen = 1;

  type_byte = TYPE_BYTE(llen, datatype);

  /* Encode and send the type.. Only the top byte is significant. */
  encodedlen = 5;
  if ((retval = ia5_encode_data(&type_byte, 1, type, &encodedlen)) != CRN_OK)
    return(retval);
  crn_put_line(type, 1);

  /* Encoded and send the length */
  encodedlen = 9;
  if ((retval = ia5_encode_data(length + (4 - llen), llen, enc_length,
                                &encodedlen)) != CRN_OK)
    return(retval);
  crn_put_line(enc_length, encodedlen);

  if (charon_debug)
    fprintf(ErrorFile, "crn_len:  %d\n", datalen);

  /* Encode the data */
  encodedlen = IA5_LENGTH(datalen);
  coded_data = (char *)malloc(encodedlen);
  if ((retval = ia5_encode_data(data, datalen, coded_data, &encodedlen))
      != CRN_OK)
    return(retval);

  if (charon_debug)
    fprintf(ErrorFile, "num_chars:  %d\n", encodedlen);
```

```
  for (count = 0; count <= encodedlen; count += LINE_SIZE) {
    if (encodedlen - count < LINE_SIZE)
      crn_put_line(coded_data+count, encodedlen % LINE_SIZE);
    else
      crn_put_line(coded_data+count, LINE_SIZE);
  }

  free(coded_data);
  return(CRN_OK);
}

/* Requires:     data be pointing into valid data, long enough to hold all the
 *               data coming in the packet, datalen should start out with the
 *               amount of space allocated, and must be a valid pointer, as
 *               datatype must also be a valid non-null pointer.
 * Modifies:     *data, *datalen, and *datatype
 * Effects:      reads data in IA5 encoding, converts it to 8-bit binary,
 *               and put it into the space of data, put the length of the
 *               data into datalen, and the packet type into datatype.
 *               Returns 0 on success, error code otherwise.
 */
int
crn_read_data(u_char *data, u_int *datalen, u_char *datatype)
{
  u_char type_byte;
  int llen, retval;
  char type[5];
  char enc_length[9];
  u_char length[4];
  u_int count, encodedlen;

  /* First, read the packet type byte (should only be one character) */
  if ((crn_read_line(type, sizeof(type))) != 1)
    return(CRN_BAD_PACKET);

  /* This is special, since I really do know the IA5 encoding scheme here */
  type[1] = type[2] = type[3] = '=';
  type[4] = '\0';

  /* Decode the type-byte, and get the llen */
  encodedlen = 4;
  type_byte = 0;
  if ((retval = ia5_decode_data(type, 4, &type_byte, &encodedlen)) != CRN_OK)
    return(retval);
  *datatype = PKT_TYPE(type_byte);
  llen = PKT_LLENGTH(type_byte);

  /* Read in the length field */
  count = crn_read_line(enc_length, sizeof(enc_length));

  /* And convert it back ... */
  BCLEAR(length);
  encodedlen = 4;
  if ((retval = ia5_decode_data(enc_length, count, length + (4 - llen),
                                &encodedlen)) != CRN_OK)
    return(retval);

  /* ... to an integer */
  octet2int(length, (int *)datalen);

  return(crn_read_packet_data(data, *datalen));
}

/* Requires:     space must be allocated already for data.  Datalen must be
 *               the length of the data.
 * Modified:     data
 * Effects:      Reads in the data in IA5 and decodes it to octet into
```

```
 *                      data.
 */
static int
crn_read_packet_data(u_char *data, u_int datalen)
{
  char *coded_data;
  u_int len, count = 0, enc_len, encodedlen;

  if (datalen == 0) {
    *data = '\0';
    return(CRN_OK);
  }

  enc_len = IA5_LENGTH(datalen);
  coded_data = (char *)malloc(enc_len--);

  while (count < enc_len) {
    len = crn_read_line(coded_data + count, enc_len - count);
    count += len;
  }

  encodedlen = datalen;
  return(ia5_decode_data(coded_data, enc_len, data, &encodedlen));
}

/* Requires:      Enough data is allocated.
 * Effects:       Read a line into data, without including CR or LF.  Returns
 *                number of characters read.  Ignore empty lines and try again.
 *                Strip off ending CR, LF, and spaces.
 */
static int
crn_read_line(char *data, int space)
{
  char buf[LINE_SIZE+3];
  char c;
  int n = 0, i = 0;

  /* No reason to read in zero data */
  if (space == 0)
    return(0);

  BCLEAR(buf);

  /* Read in the line */
  if (FGets != NULL) {
    FGets(buf, LINE_SIZE+3, InputFile);

  } else {
    if (Getc == NULL) {
      FGets = (int (*)())fgets;
      return(crn_read_line(data, space));
    }

    do {
      c = Getc();
      if (c != '\r' && c != '\n' && c != '\0') {
            buf[n++] = c;
      }
    } while (c != '\n' && n < LINE_SIZE+3);
  }

  /* Strip off trailng stuff */
  for (n = strlen(buf); n >= 0; n--)
    if (buf[n] == '\r' || buf[n] == '\n' || buf[n] == ' ')
      buf[n] = '\0';

  /* Ignore blank lines */
  if ((n = strlen(buf)) == 0)
```

```
    return(crn_read_line(data, space));

  if (charon_debug)
    fprintf(ErrorFile, "crn_get:  %s\n", buf);

  /* Copy in the known amount of space into the data provided */
  strncpy(data, buf, space);
  return(n);
}

/* Requires:      len be less than LINE_SIZE
 * Effects:       write len characters of data out, and add a CR and LF
 */
static int
crn_put_line(char *data, u_int len)
{
  char c[LINE_SIZE+3];
  int n = 0;

  if (len == 0 || strlen(data) == 0)
    return(CRN_OK);

  BCLEAR(c);
  strncpy(c, data, len);
  strcat(c, "\r\n");

  if (charon_debug)
    fprintf(ErrorFile, "crn_put:  %s", c);

  if (FPuts != NULL) {
    FPuts(c, OutputFile);
    return(CRN_OK);
  }

  if (Puts != NULL) {
    Puts(c, len + 2);
    return(CRN_OK);
  }

  if (Putc == NULL) {
    FPuts = (int(*)())fputs;
    return(crn_put_line(data, len));
  }

  for (n = 0; n < len + 2; n++) {
    Putc(c[n]);
  }

  return(CRN_OK);
}

/* Modifies:      ErrorFile, charon_debug
 * Effects:       Tries to open file for write and will set ErrorFile to
 *        that file if possible.  Also sets charon_debug if successful.
 */
void
crn_rpc_set_debug_file(char *file)
{
  ErrorFile = fopen(file, "w");
  if (ErrorFile != NULL)
    charon_debug = 1;
}
```

Appendix B

Charon Library Include Files

B.1 charon.h

```c
/* charon.h -- This file contains prototypes and definitions for
 * external programs for charon, the serial-line kerberos library
 * and extentions.
 *
 * Created by:     Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/include/RCS/charon.h,v $
 * $Author: warlord $
 *
 */

#ifndef _CHARON_H
#define _CHARON_H

#include <stdio.h>
#include <charon_err.h>
#include <des.h>

/* Sets the input file to this file */
extern void crn_set_input_file(FILE *file);

/* Sets the output file to this file */
extern void crn_set_output_file(FILE *file);

/* Sets the output character function to this function.  It expects
 * the form: int fcn(char c)
 */
extern void crn_set_putc_fcn(int (*fcn)());

/* Sets the input character function, of the form:
 * int fcn()
 */
extern void crn_set_getc_fcn(int (*fcn)());

/* Set an output-line function of the form:
 * int fcn(char *line, int len)
 */
extern void crn_set_puts_fcn(int (*fcn)());

/* Sets the output-line function, of the form:
 * int fcn(char *line, FILE *stream)
 */
extern void crn_set_fputs_fcn(int (*fcn)());
```

```
/* Sets the input-line funtion, of the form:
 * int fcn(char *data, int space, FILE *stream)
 */
extern void crn_set_fgets_fcn(int (*fcn)());

/* Server call to get tickets.  Returns CRN_OK on success, or
 * some error code.
 */
extern int crn_srv_get_tickets(char *principal, char *instance,
                               char *realm, char *hostrealm, int lifetime,
                               char *ticketfile, int version,
                               char *enc_ses, des_cblock *key);

/* Client call to get tickets.  Returns CRN_OK on success, or
 * some error code.
 */
extern int crn_cli_get_tickets(int version, int (*key_proc)(),
                               char *enc_ses, des_cblock *key);

/* Return an error message from an error number */
extern char *crn_err_msg(int error);

/* Set an error file for debugging information to go to. */
extern void crn_rpc_set_debug_file(char *file);

/* Turn echo of on the file descriptor */
extern void *crn_kill_echo(int fd);

/* Revert echoing back to its predescesor state */
extern void crn_revert_echo(void *state);

/* These definitions are for encryption using Charon.  They define how
 * the session is going to be encrypted.  There are four possibilities
 * for this value.
 */
#define CRN_ENC_NONE 0     /* No encryption */
#define CRN_ENC_CLI  1     /* Encrypt from the Client only */
#define CRN_ENC_SRV  2     /* Encrypt from the Server only */
#define CRN_ENC_BOTH 3     /* Encrypt both directions */

#endif /* _CHARON_H */
```

B.2 charon_err.h

```c
/* charon_err.h -- this file contains error codes for Charon.
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/include/RCS/charon_err.h,v $
 * $Author: warlord $
 *
 */

#ifndef _CRN_ERR_H
#define _CRN_ERR_H

#define CRN_OK                  0
#define CRN_UNKNOWN_PKT_TYPE   -1
#define CRN_ERR_DATALEN        -2
#define CRN_BAD_VERSION        -3
#define CRN_ALLOC              -4
#define CRN_BAD_PACKET         -5
#define CRN_BADPW              -6
#define CRN_KRB_PROT           -7
#define CRN_MK_PRIV            -8
#define CRN_ERR_COOKIE         -9
#define CRN_ERROR              -10


#endif /* _CRN_ERR_H */
```

B.3 charon_prot.h

```c
/* charon_prot.h -- This file contains protocol definitions for charon,
 * the kerberos extentions for secure serial operations.
 *
 * Created by:     Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/include/RCS/charon_prot.h,v $
 * $Author: warlord $
 *
 */

#ifndef _CHARON_PROT_H
#define _CHARON_PROT_H

#include <sys/types.h>
#include <netinet/in.h>
#include <krb.h>

/* A type byte is 8 bits of data, of the form:
 *         llttttuu
 *
 * where:
 *         ll is an encoding of the length of the length.
 *         ll == 0 : Llength = 1
 *            == 1 :         = 2
 *            == 2 :         = 4
 *            == 3 :         = the next byte
 *
 *         tttt is the packet type.
 *
 *         uu is unused
 */

/* This will return the value of the "ll" field in the packet type
 * byte.
 */
#define PKT_LLEN(a)    ((a & 0xC0) >> 6)

/* This will return the Length of the length (Llength).
 * a return of zero means to read the next byte
 */
#define PKT_LLENGTH(a)  ((PKT_LLEN(a) == 0) ? 1 : \
                         (PKT_LLEN(a) == 1) ? 2 : \
                         (PKT_LLEN(a) == 2) ? 4 : 0)

/* This will take a type byte and return a packet type */
#define PKT_TYPE(a)    ((a&0x3C) >> 2)

/* This will take a packet type and a Length-of-length and create
 * a type byte.  If llen != 1, 2, or 4, then there might be a problem!
 */
#define TYPE_BYTE(llen, pkt) ((((llen == 1) ? 0 : \
                                (llen == 2) ? 1 : \
                                (llen == 4) ? 2 : 3) << 6) | (pkt << 2))

/* This is the length of an IA5 line */
#define LINE_SIZE        64

/* Define the different packet types, here.  Do not change any of
 * this without also changing the table in "table.h" in the library.
 * It depends upon these number for the dispatch tables to work.
 */
#define PKT_STATUS    0          /* status information */
#define PKT_CVERSION  1          /* client version */
#define PKT_SVERSION  2          /* server version */
#define PKT_INFO      3          /* information packet */
```

```
#define PKT_SKDC        4          /* server kerberos data */
#define PKT_CKDC        5          /* client kerberos data (for interrealm) */
#define PKT_CAUTH       6          /* client authentication data */

#define MAX_NUM_PKTS 16            /* There can only be 16 packet types */

/* This is a definition to know if I am the server or the client.
 * It is not set by anything, but you know which by whether the
 * application calls the client or server function.  Do not change
 * this, as the dispatch tables depend upon this.
 */
#define CLIENT_MODE 0              /* I am a client appliation */
#define SERVER_MODE 1              /* I am a server application */

/* This defines how high our internal versions can go. Basically, it
 * is the number of protocols we have defined. It is not the same
 * as the client/server version numbers passed across in the handshake.
 * Currently, that isn't being used, but I'm allowing it for future
 * expansion of protocols.
 */
#define MAX_VERSION 2              /* Max number of versions defined in table.h */
#define MIN_VERSION 2              /* Minimum version number supported */

/* These are useful definitions for things... */
#if defined(sun) && defined(sparc) && defined(POSIX)
                                   /* solaris likes these... */
#define BCLEAR(a) memset((char *)(a), 0, sizeof(a))
#define SBCLEAR(a) memset((char *)&(a), 0, sizeof(a))
#define ZERO(a,l) memset((char *)(a), 0, (l))
#define COPY(s,d,l) memcpy((char *)(d), (char *)(s), (l))
#define CMP(a,b,l) memcmp((char *)(a), (char *)(b), (l))
#else                     /* These won't always work! (especially on solaris) */
#define BCLEAR(a) bzero((char *)(a), sizeof(a))
#define SBCLEAR(a) bzero((char *)&(a), sizeof(a))
#define ZERO(a,l) bzero((char *)(a), (l))
#define COPY(s,d,l) bcopy((char *)(s), (char *)(d), (l))
#define CMP(a,b,l) bcmp((char *)(a), (char *)(b), (l))
#endif
#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))

/* The tickets to get */
#define RCMD "rcmd"
#define KRBTGT "krbtgt"

/* This is the temp directory, for where the temp tickets should go */
#define TMP_TKT_FILE "/tmp/tkt_"

/* This is an internal structure that gets passed around inside
 * the library.  It contains a lot of information about state and
 * other interesting tidbits.
 */
typedef struct _charon_t {
 int           mode;                   /* client or server */
 int           version;                /* What version is this? */
 int           (*key_proc)();          /* Procedure to grab password */
 u_int         datalen;                /* length of that data packet */
 u_char        *data;                  /* A data packet */
 u_char        cookie[4];              /* this is a timestamp of the run */
 char          cli_encrypt;            /* Encrypt the session? */
 char          srv_encrypt;            /* Encrypt the session? */
 char          obtain_ticket;          /* Did we write a ticket to a file? */
 char          principal[ANAME_SZ];    /* principal's name */
 char          instance[INST_SZ];      /* instance name */
 char          realm[REALM_SZ];        /* principal's realm */
 char          hostinst[INST_SZ];      /* Host's kerberos name */
 char          hostrealm[REALM_SZ];    /* Realm of host */
 des_cblock    *session;               /* Session Key (if we want it) */
```

```
  struct sockaddr_in        saddr;        /* Server's IP Address */
  KTEXT         tgt;                      /* TGT ktext */
  KTEXT         tkt2;                     /* Second ticket (service or shared) */
  KTEXT         rcmd;                     /* rcmd ticket, once we have it */
} charon_t;

#endif /* _CHARON_PROT_H */
```

B.4 crn.h

```
/* crn.h -- Internal header file for the table function.  This defines
 * functions internal to crn.c
 *
 * Created by:     Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/crn.h,v $
 * $Author: warlord $
 *
 */

#ifndef _CRN_H
#define _CRN_H

#include <charon_prot.h>

static int SendVersion(int mode, int version);
static int SendVersionInfo(charon_t *charon);
static int SendInfoAndKDCPkts(charon_t *charon);
static int CheckStatus(charon_t *charon);
static int ProcessVersion(charon_t *charon);
static int ProcessInfo(charon_t *charon);
static int ProcessServerKDC(charon_t *charon);
static int DoProcessServerKDC(charon_t *charon);
static int ProcessClientKDC(charon_t *charon);
static int ProcessCliAuth(charon_t *charon);

#endif
```

B.5 ia5.h

```
/* ia5.h -- This defines the functions available in ia5.h
 *
 * Created by: Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/ia5.h,v $
 * $Author: warlord $
 *
 */

#ifndef _IA5_H
#define _IA5_H

#include <sys/types.h>

/* This will convert an length of octets to the length that the IA5
 * encoding will be, plus 1
 */
#define IA5_LENGTH(a) (((a + 2)/3)*4 + 1)


extern int ia5_encode_data(u_char *data, u_int datalen, char *encoded,
                           u_int *encodedlen);

extern int ia5_decode_data(char *data, u_int datalen, u_char *decoded,
                           u_int *decodedlen);

#endif
```

B.6 krb_prot.h

```
/* krb_prot.h -- This contains stuff that is in the prot.h file inside
 * the kerberos library.  It is neccessary because the kerberos library
 * does not the functionality I need, exactly.  I have pulled out as little
 * of the library as I needed to get my code to work.
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/athena.mit.edu/user/w/a/warlord/Thesis/src/lib/RCS/krb_prot.h,v $
 * $Author: warlord $
 *
 */

#ifndef _KRB_PROT_H
#define _KRB_PROT_H

#define KRB_PROT_VERSION    4

extern int krbONE;
#define  HOST_BYTE_ORDER     (* (char *) &krbONE)

#define pkt_version(packet)  (unsigned int) *(packet->dat)
#define pkt_msg_type(packet) (unsigned int) *(packet->dat+1)

#define pkt_a_name(packet)   (packet->dat+2)
#define pkt_a_inst(packet)   \
  (packet->dat+3+strlen((char *)pkt_a_name(packet)))
#define pkt_a_realm(packet)  \
  (pkt_a_inst(packet)+1+strlen((char *)pkt_a_inst(packet)))


#define pkt_err_code(packet) ( (char *) \
                  (packet->dat+9+strlen((char *)pkt_a_name(packet)) + \
                   strlen((char *)pkt_a_inst(packet)) + \
                   strlen((char *)pkt_a_realm(packet))))

#define         AUTH_MSG_KDC_REPLY                      2<<1
#define         AUTH_MSG_ERR_REPLY                      5<<1

#endif /* _KRB_PROT_H */
```

B.7 parser.h

```
/* parser.h -- This file contains prototypes for the procedures in parser.c
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/athena.mit.edu/user/w/a/warlord/Thesis/src/lib/RCS/parser.h,v $
 * $Author: warlord $
 *
 */

#ifndef _PARSER_H
#define _PARSER_H

#include <sys/types.h>
#include <charon_prot.h>

extern void int2octet(int num, u_char *octet);
extern void octet2int(u_char *octet, int *num);
extern int info2octet(charon_t *charon, u_char **data, u_int *datalen);
extern int octet2info(u_char *data, charon_t *charon);
extern int octet2ktext(u_char *octet, int datalen, KTEXT pkt);

#endif
```

B.8 rpc.h

```
/* rpc.h -- This contains the prototypes for functions in rpc.c
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/rpc.h,v $
 * $Author: warlord $
 *
 */

#ifndef _RPC_H
#define _RPC_H

#include <stdio.h>
#include <sys/types.h>
#include <charon_prot.h>

extern int crn_rpc_put_ktext(KTEXT pkt, int srv);
extern int crn_rpc_get_ktext(KTEXT pkt);
extern int crn_rpc_sendauth(KTEXT auth);
extern int crn_rpc_senddat(MSG_DAT *data);
extern int crn_rpc_getdat(MSG_DAT *data);
extern int crn_get_status();
extern int crn_read_status(char *msg, u_int len);
extern void crn_send_success();
extern void crn_send_error(char *msg);
extern int crn_wait_for_data(u_char **data, u_int *datalen, u_char *datatype);
extern int crn_send_data(u_char *data, u_int datalen, u_char datatype);
extern int crn_read_data(u_char *data, u_int *datalen, u_char *datatype);

#endif
```

B.9 table.h

```c
/* table.h -- this header file defines a dispatch table for Charon.
 *
 * Created by:      Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/lib/RCS/table.h,v $
 * $Author: warlord $
 *
 */

#ifndef _TABLE_H
#define _TABLE_H

#include <stdio.h>
#include <charon_prot.h>
#include "crn.h"

/* A state is an array of functions and integers, with an index of
 * a packet type.  The next_state is a number which says which state
 * it should go to next on a valid return (i.e., 0) from the function.
 */
typedef struct _state_t {
  int     (*entry)();                 /* function to call when entering this state */
  int     maxstate;                   /* Highest state in this table */
  int     (*fcn[MAX_NUM_PKTS])(); /* functions to process packet */
  int     next_state[MAX_NUM_PKTS];
} state_t;

/* A dispatch table is an array of state_t, where the index is the
 * state or the FSM.  We need one of these for each version, for
 * each side of the protcol.  There should be 16 offsets into the
 * array: one for each of the 16 possible packet types.  Use NULL
 * for an invalid packet type (error).
 */
typedef state_t *d_table;

/* A version table is an array of dispatch tables for different versions
 * of the protocol.  We need one of these for each side of the protocol.
 */
typedef d_table *v_table;

/**************************************************************/
/* Below this line is a set of definitions for the protocol.
 * It is in the form of a Finite State Machine.   Each version
 * has its own FSM, defined as an array of states.
 */

/* States for the client side of version 2 */
static state_t cv2_d_table[] = {
  {                                   /* state 0 */
    SendVersionInfo,
    3,
    {CheckStatus, NULL, ProcessVersion, NULL,
       NULL, NULL, NULL, NULL,
       NULL, NULL, NULL, NULL,
       NULL, NULL, NULL, NULL},
    {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}},

  {                                   /* state 1 */
    NULL,
    3,
    {CheckStatus, NULL, NULL, ProcessInfo,
       NULL, NULL, NULL, NULL,
       NULL, NULL, NULL, NULL,
       NULL, NULL, NULL, NULL},
    {1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}},
```

```
  {                                    /* state 2 */
    NULL,
    3,
    {CheckStatus, NULL, NULL, NULL,
      ProcessServerKDC, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL},
    {2, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}
};

/* States for the server side of version 2 */
static state_t sv2_d_table[] = {
  {                                    /* state 0 */
    NULL,
    2,
    {CheckStatus, ProcessVersion, NULL, NULL,
      NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL},
    {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}},

  {                                    /* state 1 */
    SendInfoAndKDCPkts,
    2,
    {CheckStatus, NULL, NULL, NULL,
      NULL, ProcessClientKDC, ProcessCliAuth, NULL,
      NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL},
    {1, 0, 0, 0, 0, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0}},

  {                                    /* state 2 */
    NULL,
    2,
    {CheckStatus, NULL, NULL, NULL,
      NULL, NULL, ProcessCliAuth, NULL,
      NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL},
    {2, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0}}
};

/* Client Version Table */
static d_table c_v_table[] = {NULL, NULL, cv2_d_table, NULL};

/* Server Version Table */
static d_table s_v_table[] = {NULL, NULL, sv2_d_table, NULL};

/* A command table is an array of version tables, for the client side
 * and server side of the protocol.  There is only one of these!  This
 * assumes that a client is 0, and a server is 1.
 */
static v_table command_table[] = {c_v_table, s_v_table};

#endif /* _TABLE_H */
```

Appendix C

Ckinit (src/ckinit)

C.1 ckinit.c

```
/* ckinit.c -- This source file is for ckinit, a charon initialization
 * program that uses /dev/tty as the I/O stream.  It is used to get
 * new kerberos tickets while someone is already logged in using the
 * charon protocol.
 *
 * Created by:     Derek Atkins <warlord@MIT.EDU>
 *
 * $Source: /afs/net.mit.edu/user/warlord/Thesis/src/ckinit/RCS/ckinit.c,v $
 * $Author: warlord $
 *
 */

#if !defined(lint) && !defined(SABER)
static char ckinit_c[] = "$Id: ckinit.c,v 1.9 93/05/06 01:06:38 warlord Exp $";
#endif

#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <pwd.h>
#include <krb.h>
#include <charon.h>

#if defined(sun) && defined(sparc) && defined(POSIX) /* SOLARIS */
#define ZERO(a,l) memset((char *)(a), 0, (l))
#else
#define ZERO(a,l) bzero((char *)(a), (l))
#endif

static void *tty_state = NULL;
extern char *getenv();

static void
usage(char *whoami)
{
  fprintf(stderr, "Usage:  %s [-k realm_of_host] [-p principal]\n", whoami);
  fprintf(stderr, "\t[-f ticket_file] [-l lifetime]\n");

  exit(1);
}

/* If we get a signal, we should do this, so that we can exit
```

```
 * cleanly.
 */
int
error_exit()
{
  crn_revert_echo(tty_state);
  exit(1);
}

/* I should mention that a lot of this is based off of rkinit!
 *
 * Effects:        Get kerberos tickets for a user!  Possible options to
 *                 this program include:
 *
 *        -k == set the kerberos realm of this host
 *        -p == set kerberos principal (of form princ.inst@realm)
 *        -f == ticket file to store tickets
 *        -l == ticket lifetime
 *
 *                 If no principal is passed in, then look in the ticket
 *                 file for a name.  If not there, then use username
 *                 and NULL instance, and host-realm.
 *
 *                 Use default lifetime if none is specified.
 */
main(int argc, char *argv[])
{
  char *whoami;                    /* program name. */

  char principal[MAX_K_NAME_SZ]; /* principal name */
  char a_name[ANAME_SZ];           /* These are principal, */
  char inst[INST_SZ];              /* instance, and the */
  char realm[REALM_SZ];            /* realm to use for tickets */
  char hostrealm[REALM_SZ];        /* realm of host */
  char *debug_file = NULL;         /* place to put debug info */

  char *ticketfile = NULL;         /* ticket file name */
  int lifetime = DEFAULT_TKT_LIFE; /* ticket lifetime */
  int i, status = 0;
  struct passwd *pwd;              /* local passwd info */

  /* Clear out these buffers */
  ZERO(principal, sizeof(principal));
  ZERO(a_name, sizeof(a_name));
  ZERO(inst, sizeof(inst));
  ZERO(realm, sizeof(realm));
  ZERO(realm, sizeof(hostrealm));

  /* Set the program name. */
  if ((whoami = strrchr(argv[0], '/')) == NULL)
    whoami = argv[0];
  else
    whoami++;

  /* Deal with the command-line options */
  for (i = 1; i < argc; i++) {

    /* make sure that its an option! */
    if (argv[i][0] != '-')
      usage(whoami);

    switch(argv[i][1]) {

    case 'k':                      /* realm */
      if (++i >= argc)
            usage(whoami);
      strncpy(hostrealm, argv[i], sizeof(hostrealm) -1);
      break;
```

```
  case 'p':                        /* principal */
    if (++i >= argc)
          usage(whoami);
    strncpy(principal, argv[i], sizeof(principal) -1);
    break;

  case 'f':                        /* ticket file */
    if (++i >= argc)
          usage(whoami);
    ticketfile = argv[i];
    break;

  case 'l':                        /* lifetime */
    if (++i >= argc)
          usage(whoami);
    lifetime = atoi(argv[i])/5;
    if (lifetime = 0)
          lifetime = 1;
    else if (lifetime > 255)
          lifetime = 255;
    break;

  case 'd':                        /* debug */
    if (++i >= argc)
          usage(whoami);
    debug_file = argv[i];
    break;

  default:
    usage(whoami);
  } /* switch */
} /* for */

if ((pwd = getpwuid(getuid())) == NULL) {
  fprintf(stderr, "%s:  Cannot find your entry in the password file.\n");
  exit(1);
}

/* Make sure we have an a_name, inst, and realm to pass in.
 * (we really only need an a_name, since the library will deal
 * with everything else.
 */
if (! principal[0]) {

  /* No principal passed in -- look in ticket file */
  if ((status = krb_get_tf_fullname(TKT_FILE, a_name, inst, realm))
    != KSUCCESS) {
    /* No ticket file.  use username@hostrealm */
    strncpy(a_name, pwd->pw_name, sizeof(a_name) - 1);
    strcpy(realm, hostrealm);
  }
} else {
  if ((status = kname_parse(a_name, inst, realm, principal))
          != KSUCCESS) {
    fprintf(stderr, "%s:  %s\n", whoami, krb_err_txt[status]);
    exit(1);
  }
}

tty_state = crn_kill_echo(0);

/* trap a lot of signals */
signal(SIGHUP, error_exit);
signal(SIGINT, error_exit);
signal(SIGQUIT, error_exit);
signal(SIGKILL, error_exit);
signal(SIGTERM, error_exit);
```

```
  if (debug_file != NULL)
    crn_rpc_set_debug_file(debug_file);

  status = crn_srv_get_tickets(a_name, inst, realm, hostrealm,
                               lifetime, ticketfile, 0, 0, 0);

  crn_revert_echo(tty_state);

  return(status);
}
```

Bibliography

[1] E. J. Berkenbilt. Rkinit. Personal Communiqué, November 1989.

[2] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Usenix Conference Proceedings*, pages 191–202, Dallas, Texas, February, 1988.

[3] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC 1421, February 1993.

[4] National Bureau of Standards, Washington, D.C. *Data Encryption Standard*, Federal Information Processing Standards Publication 46 edition, 1977. Government Printing Office.

[5] B. Clifford Neuman and Jennifer G. Steiner. Authentication of Unknown Entities on an Insecure Network of Untrusted Workstations. In *Usenix UNIX Security Workshop Proceedings*, pages 10–11, Portlan, Oregon, August, 1988.

[6] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos Authentication and Authorization System. Technical report, M.I.T. Project Athena, "1985, 1986, 1987".