Final thesis

Reverse-Engineering and Implementation of the RDP 5 Protocol

by

Erik Forsberg

LiTH-IDA-EX--04/082--SE

2004-03-05

Final thesis

Reverse-Engineering and Implementation of the RDP 5 Protocol

by

Erik Forsberg

LiTH-IDA-EX-04-082--SE

2004-03-05

Supervisors: **Peter Åstrand**
Thomas Rosén

at Cendio AB

Examiner: **Dr. Juha Takkinen**
Department of Computer and Information Science
at Linköpings universitet

Abstract

The Remote Desktop Protocol (RDP) is a protocol for accessing Microsoft Windows Terminal Services. The protocol provides remote desktop services, meaning a graphical desktop is sent to the client, and user input (keyboard and mouse events) are sent to the server, all over a bandwidth-narrow channel. The protocol is used by thin clients, i.e. clients with small resources, to reach servers in a server-based computing environment.

There is an RDP-client called Rdesktop, written for Unix-like operating systems. It has an X Window System graphical user interface and provides access to Terminal Servers from the Unix environment. Rdesktop, however, only supports version 4 of the RDP. The current version of RDP (August 2003) is 5.

Documentation of RDP can be acquired from Microsoft, but not without signing a non-disclosure agreement (most often referred to as “NDA”). This means it is not possible to create a program with the source code available without breaking the agreement. Therefore, implementation of open source RDP clients must be preceded by reverse engineering activities.

In this report we describe how we reverse engineered version 5 of RDP and how we implemented support for it in rdesktop. We have implemented support for basic RDP5 as well as support for clipboard operations between the X Window System and Microsoft Windows.

Among the future work on rdesktop that will be possible to investigate as a result of this thesis work are support for sound redirection, disk drive redirection as well as support for more clipboard formats.

Keywords: Reverse Engineering, Network Analysis, Software Engineering, Network Security, Remote Desktop Protocol.

Acknowledgments

During the work with this final thesis, we have had great help from a number of people. I would like to thank the following persons:

- Peter Åstrand, supervisor at Cendio
- Thomas Rosén, supervisor at Cendio
- Juha Takkinen, my examiner at the Department of Computer and Information Science, Linköpings universitet
- Matt Chapman, Jay Sorg, Peter Byström, and the rest of the rdesktop team.

Contents

Chapter 1

Introduction

1.1 Background

The Remote Desktop Protocol (RDP) [Mic04a] is used to access Microsoft Windows Terminal Services [Mic04e] and provide remote desktop services using a network channel with narrow bandwidth. Remote desktop services are the ability to run programs on the server and get the output on the local screen, sending keyboard and mouse events to the server.

"Official clients", that is, clients written by or approved by Microsoft, for RDP exist for Microsoft Windows [Mic04d] and Mac OS X [Mic04c]. There are also some commercial clients available for Unix-like environment [Ent04], but there is only one client that is open source software, namely rdesktop [ini03], written for the X Window System [Fou04b]. Since the RDP is a closed protocol, parts of it have been reverse-engineered in order to find the information needed to write rdesktop.

Rdesktop is used in ThinLinc, the main product of Cendio AB, to access Windows Terminal Servers in setups that need that kind of access. This is the reason why Cendio AB is sponsoring this thesis work.

1.2 Motivation

There are one primary and one secondary reason motivating this final thesis work.

- **Primary Reason - Rdesktop Needs Enhancements**

  The primary reason for this thesis work is that rdesktop needs to be enhanced with functionality that exists only in version 5 and higher of the RDP, the version of the Remote Desktop Protocol used in Microsoft Windows 2000 and newer.

- **Secondary Reason - Documentation Needed for Further Development of Rdesktop**

  The secondary reason is that the documentation on RDP and internal rdesktop functionality is very poor. We want to make it easier to add functionality to rdesktop, by providing better documentation.

1.3 Problems to be Solved and Goals of Thesis

The specific problem to solve in this final thesis is to add clipboard functionality to rdesktop, allowing cut'n paste operations from Windows programs to X Window System programs, and vice versa. At the end of this thesis work, we expect to be able to cut text in an instance of Microsoft Word [Mic04b] running on a Windows Terminal Server, and paste it into an instance of StarOffice Writer [Mic04f] running on a Linux host.

In order to solve this problem, we will investigate how version 5 of the RDP works. We need to do this because Clipboard functionality exists only in RDP version 5 and higher. When done with the investigation, we will add RDP5 support to Rdesktop and continue by investigating how clipboard data is transmitted over RDP5. When we have gathered enough information, we will add clipboard functionality to rdesktop.

With regard to the poor documentation, we expect this report to be helpful in future development of rdesktop.

1.4 Outline of the Report

In chapter 2 we describe the concepts behind reverse-engineering network protocols, and the methods we use to find the information we need in order to implement RDP5 and clipboard support in rdesktop. The information found using these techniques and also by reading various specification documents are documented in chapter 3, where we cover how the Remote Desktop Protocol is constructed. Chapter 4 covers how we implement RDP5 and clipboard support, including interoperability problems between the different clipboard paradigms used by the two platforms involved. Finally, we draw conclusions and discuss possible extensions in chapter 5. As a service for future developers, we have written appendix A where we describe the different packets used in the protocol.

1.5 Target Audience

This report assumes basic knowledge of computer network protocols, computer security and software implementation issues. Some understanding of how Windows Terminal Services [Mic04e] work will also help for full comprehension of the report. Parts of chapter 4 are of use only if the reader plans to do changes to rdesktop. Appendix A is of use only for the reader that wants a deeper understanding of the packet structures used in RDP.

1.6 Typographical Conventions and Nomenclature

References to sourcecode filenames in this report are written in typewriter style, like `rdesktop.c`. Datastructures and variables are in italic style, like *int testval*.

In this document, we use RDP4 as a shorthand notation for Remote Desktop Protocol version 4 and RDP5 for Remote Desktop Protocol version 5. RDP is used when we speak about the protocol in general.

Chapter 2

Reverse-Engineering Network Protocols

In this chapter we discuss reverse-engineering of network protocols. Some background to the need of reverse-engineering is given, as well as some political aspects. The problems of planning reverse-engineering activities are briefly mentioned.

Finally, we discuss the methods we will use to find the information needed to describe the protocols and implement support for RDP5 in rdesktop.

2.1 Introduction

2.1.1 Concepts

The concepts used in the reverse-engineering field seem not to be standardized. Therefore, **our** definitions of some terms follow.

- **Reverse Engineering**
  Depending on whom you ask, the term Reverse Engineering is used for different things. In the academic world, the term is used to describe

the problem of getting information out of large amounts of source code, as a prelude to some software maintenance activity (see section 4.1.1). One example of this use of the term can be found in [MJS+00].

However, in other parts of the computer industry, the term is used in a more general way, describing activities that in some way are used to find unpublished information. This is how we use the term.

- **Network Analysis**
  Network analysis is another term used in technical network communities. It describes methods and tools for analyzing raw network data in order to find problems with a network, for example bottlenecks and intrusions.

Another related term is “Retro engineering”, which is used when there is a need to rewrite a standard in order to comply with already existing and market-leading incorrect implementations. This has happened to for example X.509 [Int97].

2.1.2 The Need for Reverse Engineering

Why is there a need for reverse-engineering methods? Some common answers to this question are:

1. The source code of a piece of software might have been lost accidentally, but new software needs to be written that uses the same protocols for its communication.
2. The people that first wrote the software are not available at the moment, but the software needs to be modified by some reason. This is part of the academic definition of reverse engineering.
3. Legacy data exists in a format that no existing software can parse.
4. There is no suitable software available for a client platform that needs to communicate with some specific server application.

The task in this final thesis is of the last kind. There is a need to communicate with Windows Terminal Services from Unix-like platforms, but there is no client available. Since the protocol is only partially documented outside Microsoft, reverse-engineering is needed for full functionality.

2.2 Political and Legal Aspects

This project had to take some non-technical matters into account. In this section we will discuss the most important of them briefly.

2.2.1 Why Keeping a Network Protocol Secret?

As we will describe in chapter 3, the protocols used in rdesktop are only partially documented. Here we discuss pros and cons of public protocols.

If a company has a truly superior protocol for a specific task, they may choose to keep it secret in order to gain market shares while not leaking the results of their research and development to their competitors.

However, the problem with this approach is that you completely lose compatibility with existing products, something that is becoming more and more important with time, as the computing environments grow in complexity.

Also, it is very seldom possible to keep the protocol secret for a long time, since reverse-engineering methods do exist. See section 2.3 for more information.

Another reason for keeping the protocol secret, even though it is not superior in any way, is as part of a larger plan to actually increase the incompatibility with other products. By creating a set of products that only communicate with each other, a large company can create it is own market where the customer is forced to buy more products from the same vendor when a certain functionality is needed, since the functionality will not integrate as well if another vendors products are used. The plan may not always work, since by using reverse engineering, other companies, organizations or individuals can create products that do communicate with the large company's products, even though the protocols were secret. An example of this is the SMB file server Samba [dev03], providing Windows file server services using a Unix machine as server.

Yet another reason for keeping a protocol secret can be security. Some people believe security is gained by keeping protocols closed. This principle is often referred to as “Security by Obscurity”. In [CP98], the authors come to the conclusion that “Security by Obscurity” is often useless since the attacker quickly learns how the obscured system works. Other people believe a security-sensitive protocol that cannot stand the eyes of the world is not secure. The latter opinion is often found in Internet and Open Source communities.

2.2.2 Legal Aspects

Various laws exist that either forbid or allow reverse engineering.

Since this final thesis work is written in Sweden, which is part of the European Community, we have only investigated the laws governing reverse-engineering in the EC. The situation in the USA is another subject.

Most relevant to our project is the “COUNCIL DIRECTIVE of 14 May 1991 on the legal protection of computer programs (91/250/EEC)” [cotEc]. As far as we can see as laymen in law interpretation this law permits us to reverse-engineer the network protocols used in RDP. However, each party interested in doing some kind of reverse-engineering should investigate the relevant laws at that time.

2.3 Methods for Reverse Engineering

A number of common methods for reverse-engineering exists. We will describe some of them briefly. During our work, we have developed some methods for our specific task that we will describe as well.

2.3.1 Common Reverse-Engineering Methods

Disassembler and Debugger

One way of finding out exactly what a piece of software is doing is to examine the binary executable machine code. Given the machine instructions of a certain hardware platform it is possible to decode the machine code into assembler instructions using a so called disassembler [Wik05b].

However, since the assembler instructions are most often generated by a compiler from some higher level language, you will get a very hard-to-read representation of what the program does.

It is often possible to debug the program at an assembler code level in order to find the correct lines of code in the program for your specific task.

However, this method is very time-consuming and tedious. It is not practical for larger projects, but for small well-defined tasks it can be an effective method. An example would be the task to find out how a certain image format is encoded by running an image decoding program to see what it is doing for a certain image that has a known content.

Man In The Middle Attack

When doing reverse-engineering of network protocols, a method known as "Man In The Middle attack" [Wik05c], hereafter referred to as **MITM attack**, can be very useful.

In the MITM attack, the network path between two communicating computers is by some method altered in order to route the network packets through a third computer where a specially written piece of software handles the packets. In figure 2.1, the normal path for packets that travel between the client and the server is *Network path A*.

In order to use the MITM attack, we tell the client it should connect to the computer running the MITM software instead of connecting to the real server. That is, the packets will travel *Network path C*. When the packets arrive to the MITM computer, they are processed by the MITM software and then sent via *Network path B* to the server. Packets from the server are handled the same way, but in the opposite direction.

The MITM software can either just listen to the packets (although a network packet sniffer (see section 2.3.1) can do that job just as well or even better) or process them in a more advanced fashion, for example by replacing some part of the packet in order to instruct either the client or the server to behave in a certain way.

There are several different types of MITM software. Either the software is very simple and does nothing else than listening for packets, presenting them without modification to the party operating the MITM software. For this purpose, a network packet sniffer, as described below, can probably

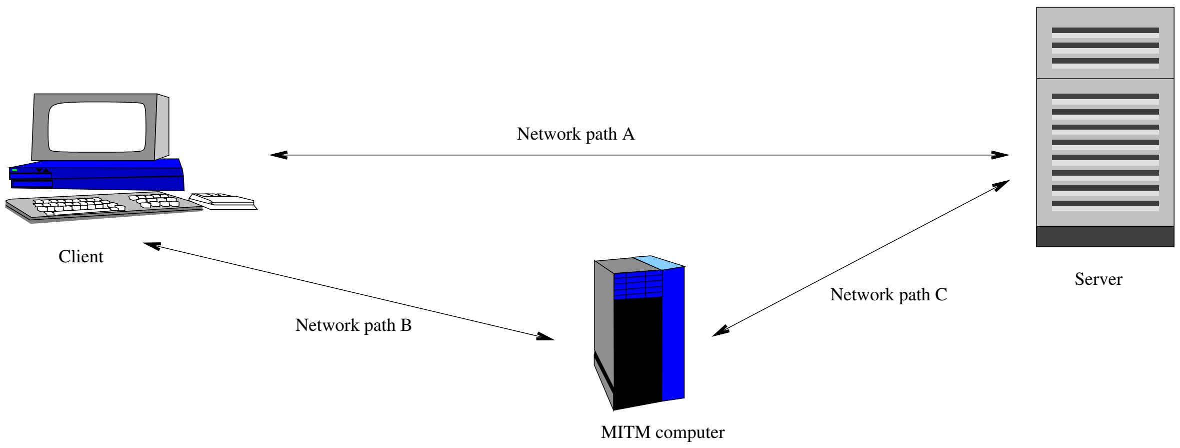

Figure 2.1: Man In The Middle attack

do the job just as well. The MITM software can also do more advanced processing of the packages. It can, for example, replace parts of the packet on its way, in order to instruct either the client or the server to behave in a certain way. A real-world example of this was when the Samba [dev03] project reverse-engineered part of the authorization protocols by using an MITM software that replaced parts of the network packet stream in order to force the native servers to talk to each other in a different way [Tri03].

Network Packet Sniffer

Another very useful tool when reverse-engineering network protocols is a packet sniffer. A packet sniffer works by listening to the traffic of a network interface and presenting the packets in a partially or fully parsed fashion. Examples of such software are tcpdump [Lab04], snoop [Mic] and Ethereal [Com04].

Depending on the network architecture, the packet sniffer can only listen to packets where the host it is running on is the destination or source, or on all hosts on the same broadcast domain.

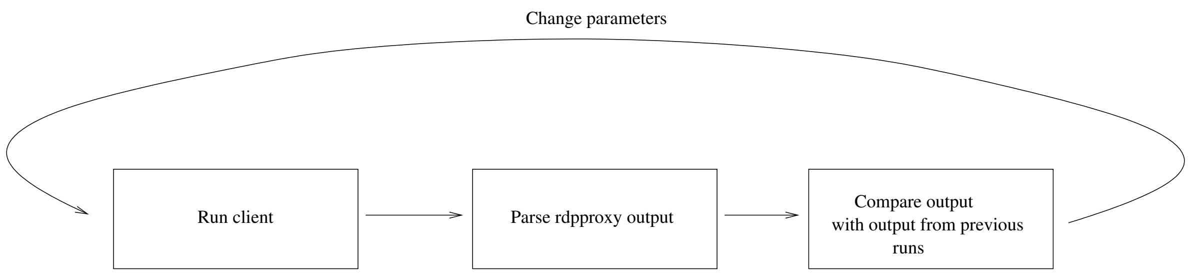

Figure 2.2: The iterative reverse-engineering process

2.3.2 Our Approach for Reverse Engineering RDP

An MITM attack is used in our task, for a motivation see section 2.3.3. Our solution is of the more advanced type since we replace some bits in the protocol in order to be able to read it in plaintext. That is, our proxy modifies the data stream from the client to the server.

Finding the information needed for a full understanding of the protocol is an iterative process with the following steps:

1. Sniff and save the network traffic between client and server using rdpproxy (rdpproxy is our MITM software, see section 2.3.3).
2. Parse the network traffic using `pparser.py` (see section 2.3.3).
3. Analyze output from `pparser.py`. Modify `pparser.py` and rerun it when finding new information.
4. Compare output from previous runs with this one.
5. Modify some parameter on the client (colour depth, for example)
6. Start again at step 1. In some cases after modifying rdpproxy.

The process is also illustrated in figure 2.2.

For example, given the qualified guess that the server would send a cryptographic salt (a piece of randomness used during crypto protocol setup) that would differ between two sessions (in fact, it should be different for all sessions if it is cryptographically safe), by comparing the network packets sent from the server in two different sessions, we could find where and how the server salt was sent, which was information we needed in order to find out how to decrypt the session.

2.3.3 Reverse Engineering Tools Used and Developed

When we began investigating the existing implementation (rdesktop) and the protocol we saw several subproblems:

- The protocol is only partially specified in documentation available outside Microsoft, which means we needed to develop a way to see the contents of the packets the parts in the network session exchange.
- The protocol is encrypted which means a packet sniffer will only see random data.
- Possibly we would need to modify the data stream in order to understand the protocol.

This quickly led us to the conclusion that an MITM attack would be a good way to retrieve the data we needed.

We decided against using a debugger and/or a disassembler for several reasons. First, our knowledge of network sniffing is much larger than our knowledge of debuggers and disassemblers running on Windows. Secondly, the company kindly hosting this final thesis work has a computer environment with much more Linux than Windows computers, meaning a solution running without modifying the few Windows computers fits much better into the environment. Also, debuggers and disassemblers for Windows most often are commercial software, and the budget for this work is limited.

The fact that the author of rdesktop (Matt Chapman) already had developed the skeleton of an MITM software also helped, both by confirming our decision and by providing us with a start in the process.

Rdpproxy

The MITM software that the rdesktop author had started writing when we began our work is named “rdpproxy” [FC04] since it is “proxying”, in the meaning "forwarding traffic in both directions", RDP. We have run rdpproxy on Linux systems, although it should run without problems on any unix and maybe on Windows.

The version we began with basically works as follows:

1. Rdpproxy is started on the MITM computer (see figure 2.1) with the server name as a command line argument.
2. Rdpproxy listens to the RDP port (TCP port 3389).
3. When the client computer connects to the MITM computer, rdpproxy connects to the server specified on its command line.
4. In the encryption setup the public key of the server is replaced by another public key for which rdpproxy knows the private key. When the client sends back its encrypted salt, rdpproxy decrypts it with its private key and then reencrypts it with the public key sent from the server. The client salt is stored and used together with the server salt to calculate the session key.
5. Packets from the client are forwarded to the server.
6. Packets from the server are forwarded to the client.
7. All packets are printed as hexadecimal dumps to the standard output, in plaintext since we do have the private session key which we retrieved earlier in step 4.

We began by removing some hardcoded assumptions in the software. This made rdpproxy work well with servers in *Remote Administration Mode*. We still had some trouble with servers in *Terminal Server Mode* due to the encrypted licensing negotiation. See section B.3 for a description of the different server modes available. Since we could choose what mode the server should use, the licensing problems was not an issue. We get an unencrypted transcript of what the client and the server is saying to eachother, and that is what we need.

Later, we modified rdpproxy in order to support *Terminal Server Mode* as well by modifying the decryption and encryption sequence for traffic from the client to the server.

```
RDPPROXY: waiting for connection...
#1, #1 from Client, type TPKT, l: 40, read 40 bytes
Client key substitution not done..
Trying to substitute crypt type..
Cannot decrypt, haven't seen client random!
0000 03 00 00 28 23 e0 00 00 00 00 00 43 6f 6f 6b 69 ...(#......Cooki
0010 65 3a 20 6d 73 74 73 68 61 73 68 3d 61 64 6d 69 e: mstshash=admi
0020 6e 69 73 74 72 61 0d 0a                         nistra..
#2, #1 from Server, type TPKT, l: 11, read 11 bytes
0000 03 00 00 0b 06 d0 00 00 12 34 00                .........4.
#3, #2 from Client, type TPKT, l: 412, read 412 bytes
Client key substitution not done..
Trying to substitute crypt type..
Substituted crypt type
Cannot decrypt, haven't seen client random!
0000 03 00 01 9c 02 f0 80 7f 65 82 01 90 04 01 01 04 ........e.......
0010 01 01 01 01 ff 30 19 02 01 22 02 01 02 02 01 00 .....0..."......
0020 02 01 01 02 01 00 02 01 01 02 02 ff ff 02 01 02 ................
0030 30 19 02 01 01 02 01 01 02 01 01 02 01 01 02 01 0...............
0040 00 02 01 01 02 02 04 20 02 01 02 30 1c 02 02 ff ....... ...0....
0050 ff 02 02 fc 17 02 02 ff ff 02 01 01 02 01 00 02 ................
0060 01 01 02 02 ff ff 02 01 02 04 82 01 2f 00 05 00 ............/...
0070 14 7c 00 01 81 26 00 08 00 10 00 01 c0 00 44 75 .|...&........Du
0080 63 61 81 18 01 c0 d4 00 04 00 08 00 20 03 58 02 ca.......... .X.
0090 01 ca 03 aa 1d 04 00 00 28 0a 00 00 53 00 41 00 ........(...S.A.
00a0 55 00 52 00 4f 00 4e 00 53 00 5f 00 54 00 4f 00 U.R.O.N.S._.T.O.
00b0 49 00 4c 00 45 00 54 00 00 00 00 00 04 00 00 00 I.L.E.T.........
00c0 00 00 00 00 0c 00 00 00 00 00 00 00 00 00 00 00 ................
00d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0100 00 00 00 00 00 00 00 00 01 ca 01 00 00 00 00 00 ................
0110 08 00 07 00 01 00 00 00 00 00 00 00 00 00 00 00 ................
0120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0150 00 00 00 00 00 00 00 00 04 c0 0c 00 09 00 00 00 ................
0160 00 00 00 00 02 c0 0c 00 01 00 00 00 00 00 00 00 ................
0170 03 c0 2c 00 03 00 00 00 72 64 70 64 72 00 00 00 ..,.....rdpdr...
0180 00 00 80 80 63 6c 69 70 72 64 72 00 00 00 a0 c0 ....cliprdr.....
0190 72 64 70 73 6e 64 00 00 00 00 00 c0             rdpsnd......
```

Figure 2.3: Sample output from rdpproxy

As we can see in figure 2.3 the output from rdpproxy is quite raw. There is some help for the human reader to understand the format: The number of the packet and which part it originates from is printed, and there is an offset that marks where in the packet a certain line is. Also, there is a textual representation to the right which prints bytes that can be interpreted as characters as the decoded characters which helps in understanding the text strings in the protocol.

Understanding the Output of Rdpproxy with pparser.py

During our early efforts to understand the process we tried to hand-parse some of the packets in the output from rdpproxy. Although it is certainly possible to understand the output of rdpproxy by hand, it is a time-consuming and error-prone process, not to mention it is extremely boring. If we would have had to hand-parse the packets each time we have made a change in the configuration, a lot of time would have been wasted just on reading hexdumps. We quickly decided we needed a better method to get a human-readable output from rdpproxy. As we describe in section 2.3.2, we quickly found we would have to analyze and compare a lot of different rdpproxy outputs.

We investigated several possible ways before deciding how to create a tool that would parse the output from rdpproxy:

- Reusing code from rdesktop was one option, since rdesktop already has an RDP parser. This fell on the fact that rdesktop is written in C which means it takes time to modify and recompile. Also, C is very error-prone compared to other languages, since you have to do all error handling yourself. While implementing parts of the RDP5 code we found this very true. It took many times more time to implement for example the logon packet parser in C than it took to parse it using `pparser.py`.

- Building a plugin that would fit into Ethereal was another option. Ethereal is a packetsniffer (see section 2.3.1) with a really userfriendly graphical user interface. It can use plugins that parse the packets and present the different packet parts in a hierarchical tree which makes it easy to understand the structure. However, we decided against this alternative for several reasons:

  – Documentation on how to write plugins for Ethereal was hard to find.
  – We needed a way to provide the output from rdpproxy in a format that Ethereal could read. One option was to write a libpcap file, but documentation on how to do this was hard to find as well.
  – We wanted real-time parsing, where we could see what happened on the network in the same moment it happened. For example, while finding out how clipboard worked, we wanted to do a clipboard operation and at the same time see what channel was used and what data was transferred at which moment. We did not find any method to connect rdpproxy and Ethereal in a way that would accomplish this goal.
  – Documentation available stated that plugins must be written in C, something we wanted to avoid for the same reasons we chose not to use the available RDP parser in rdesktop.

- Building a new packet parser in a less error-prone language than C. This was the option we chose. Building a packet parser that could understand the output from rdpproxy enabled us to concentrate on the contents of the packets, not how to convert them from one format to another before trying to analyze them.

We built `pparser.py`, an RDP packet parser written in Python [Fou03]. We wrote it in Python since we know from previous experience that we write textparsing applications very quickly in that language. Also, it has all the features we wanted like object-orientation, extremely good textparsing abilities and automatic memory handling. Since we knew we would have to do a lot of modifications in order to test different theories, we wanted an interpreting language which gives a short development cycle. Quoting [Pre00]: "Designing and writing the program in Perl, Python, Rexx, or TCL takes only about half as much time as writing it in C, C++, or Java and the resulting program is only half as long". Another factor that helped in the choice of language was that Python is extensively used at Cendio, meaning local support would be available.

The parser is run from the command line and reads one input file of the format rdpproxy outputs and writes one output file in one of serveral textbased formats. It can read from standard input and write to standard output if desirable, which enables the real-time parsing we wanted by standard unix output redirection methods (shell pipes).

A sample of the text format `pparser.py` outputs can be seen in figure 2.4. Note that this is the result of running pparser on parts of the rdpproxy output in figure 2.3. The third packet has been left out for spacesaving reasons. See appendix A for a complete documentation of packet contents.

The output format is still not understandable without knowledge about the protocol, but it is much easier. The real advantage of the format compared to the raw hexadecimal dump produced by rdpproxy is that when we started comparing two or more sessions using methods described in section 2.3.3, we did not have to spend time finding the correct bytes to compare. The format produced by `pparser.py` is much easier to navigate, since each interesting piece of data get its own line instead of being hidden together with up to 15 other bytes on one line.

Another positive effect of building the packet parser was the fact that it contains implicit documentation of the network protocol in a quite readable way. We modified the parser to output LaTeX [Pro05] tables that we include in this report. See appendix A for the output produced this way. An advantage of this is that when we found errors in the way we parsed the packets and modified the packet parser, this report would concur with the new version automatically, as long as we regenerated the tables, an automated process.

Finding Differences Between Different Sessions

As we mentioned in section 2.3.2, we compared the data sent between the two parties in the session in different sessions in order to find out for example what had changed when we changed some parameter. We used several tools to do this:

- **diff**

```
Unknown data: RDPPROXY: waiting for connection...
#1, #1 from Client, type TPKT, l: 40, read 40 bytes
TPKT from Client
 Int8 (be) TPKT version (expected: 3) 0x03 (3)
 Int8 (be) TPKT reserved (expected: 0) 0x00 (0)
 Int16 (be) TPKT length 0x0028 (40)
 ISO packet
   Int8 (be) ISO hdr length 0x23 (35)
   Int8 (be) ISO packet type 0xe0 (224) Connection request
   Int16 (be) Dst ref 0x0000 (0)
   Int16 (be) Src ref 0x0000 (0)
   Int8 (be) Class 0x00 (0)
[unknown type] Remaining data  RAW DATA (length 0x1d (29))
    43 6f 6f 6b 69 65 3a 20 6d 73 74 73 68 61 73 68 Cookie: mstshash
    3d 61 64 6d 69 6e 69 73 74 72 61 0d 0a          =administra..
#2, #1 from Server, type TPKT, l: 11, read 11 bytes
TPKT from Server
 Int8 (be) TPKT version (expected: 3) 0x03 (3)
 Int8 (be) TPKT reserved (expected: 0) 0x00 (0)
 Int16 (be) TPKT length 0x000b (11)
 ISO packet
   Int8 (be) ISO hdr length 0x06 (6)
   Int8 (be) ISO packet type 0xd0 (208) Connection confirm
   Int16 (be) Dst ref 0x0000 (0)
   Int16 (be) Src ref 0x1234 (4660)
   Int8 (be) Class 0x00 (0)
```

Figure 2.4: Sample output from pparser.py

The standard unix command **diff** was used to compare output from both rdpproxy and the parsed output from `pparser.py`. However, the output from **diff** is sometimes hard to follow.

- **ediff**
  We found that the **ediff** package in the Emacs [Fou04a] editor was a very efficient way of finding differences in the files we were looking at. Ediff is a frontend to the diff command that visualizes the differences by showing both files at the same time and highlighting the differences in a way that is very easy to follow.

  Ediff can also calculate the differences between three different files, something we sometimes did in order to find what was supposed to change between sessions (cryptographic salts and signatures) and what had changed because of some parameter change we did as part of the iterative process (section 2.3.2).

2.4 Planning and Keeping Track of Data

Planning and time-estimating reverse-engineering activities are difficult. Most projects in this area depend on some kind of breakthrough, and to know when this breakthrough will occur is very hard.

While waiting for the breakthrough, both patience and a structured work flow are of great importance. Keeping track of all information gained through the process is essential since you do not know what small bits of information might be the key to the problem.

We selected to use a simple diary for storing the information we found during the process. We developed a very simple snippet of lisp code for the Emacs editor that makes it possible to press a quick key combination and just start to write whatever is in our minds at each moment. One file for each day is created, with timestamps for each note, and a simple shell script makes it possible to create a summary of the diary. We expected this simple tool to be of great help in our work.

Chapter 3

The Protocols Involved in Terminal Services

In this chapter, we will discuss how RDP works as a network protocol. We will begin by describing the protocol in general, to give an overview for newcomers to RDP. We will however concentrate on the differences between RDP version 4 and 5. This is beacuse the primary goal of this thesis is to implement RDP5 support in rdesktop, and rdesktop already has support for RDP version 4.

Apart from discussing and describing RDP, we will also discuss network security and security-related protocols.

3.1 The RDP Network Stack

Similarly to most network protocols, RDP consists of several layers. Figure 3.1 gives an overview of how RDP4 is built. There are some differences in RDP5, which we will describe later. For an in-depth reference of the different packets involved in RDP, see appendix A.

Beginning from the bottom of the stack, there is a standard TCP [STD81] connection from the client to port 3389 on the server.

On top of TCP, ISO DP 8073 packets are sent. This is a standard

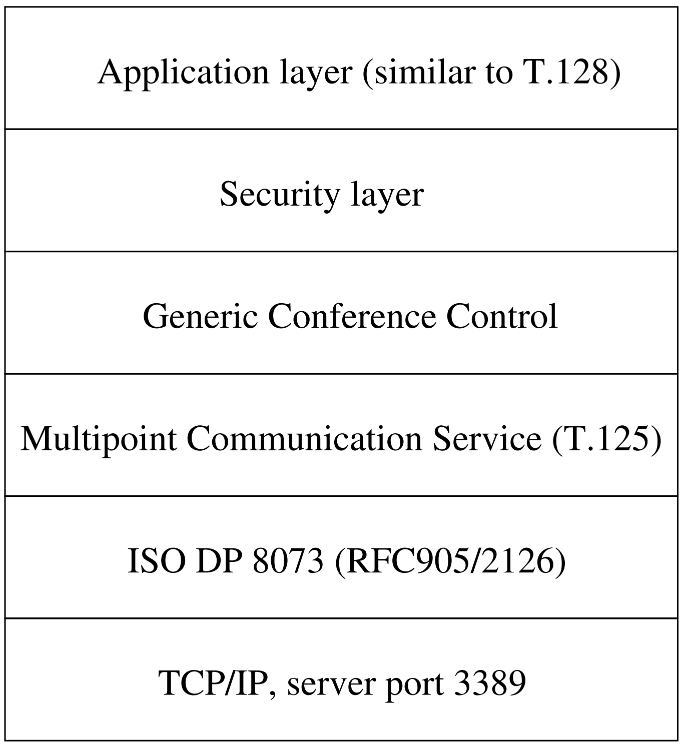
Application layer (similar to T.128)
Security layer
Generic Conference Control
Multipoint Communication Service (T.125)
ISO DP 8073 (RFC905/2126)
TCP/IP, server port 3389

Figure 3.1: The Remote Desktop Protocol, version 4

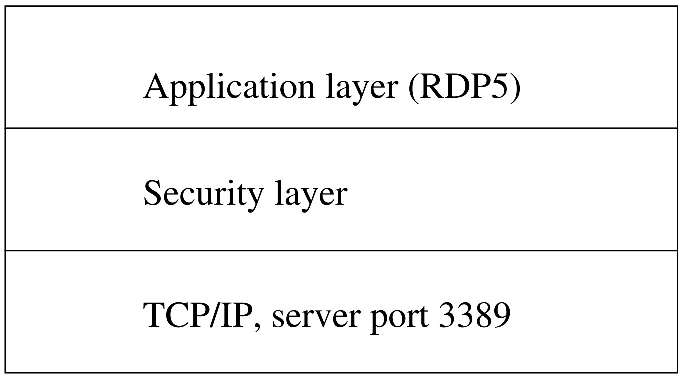

Figure 3.2: The Remote Desktop Protocol, version 5

defined in [PY97], meant to provide a way for ISO network standards (built on the well-known OSI model, see [Tan02] p. 37 ff) to run on top of the more commonly implemented TCP. This protocol is also referred to as "TPKT", for example in the network analysis program Ethereal [Com04].

On top of ISO DP 8073 lies Multipoint Communication Services (MCS), defined in [T.193b]. This is a standard providing domain management, channel management, data transfer, and token management. The latter is not used in RDP. The domain management is sparsely used too, since it is meant for protocols where more than two entities are communicating with each other. The channel management is somehow used in RDP4, but there is only one channel in use. In RDP5, more channels are used for example for sending clipboard and sound data. The data transfer capability is used to send the data between client and server.

The Generic Conference Control (T.124 [T.193a]) is almost invisible in the protocol implementation. There is a minor exchange of GCC data in each direction in the protocol setup phase, but its role is so small we will not mention it further. Rdesktop implements GCC by sending a hardcoded string to the server, and by skipping 21 bytes when reading one of the setup packets from the server.

The security layer provides encryption of the data sent over the MCS. The encryption algorithms used is covered in section 3.3.

On top of the protocol stack, the RDP application layer is the protocol that defines how the graphic data is sent to the client, and how mouse and keyboard data is sent to the server. This protocol is based on [MPP97], but Microsoft has redefined it somewhat since then, especially in RDP5. We will cover the changes later.

3.2 Changes in RDP5

In version 5 of the Remote Desktop Protocol, Microsoft has redesigned parts of the protocol. Since we have no access to the design criteria of Microsoft, we can only speculate in the design goals for this new version of the protocol. Most probably, bandwidth savings were part of the specification, but performance might have been on the list as well, since the server now has to do less processing per packet, which means it can handle more simultaneous connections.

Figure 3.2 gives an overview of the new protocol. The difference between RDP4 and RDP5 is that the ISO DP 8073, MCS, and GCC layers are stripped away. Instead, a new more compact packet format is defined. For a detailed view of an RDP5 packet, see section A.3.1.

Not all packets are of RDP5-type. The setup phase is exactly the same as before, and all packets from the client to the server are still of the same format as in RDP4. RDP5 packets are used only for graphic data from the server. Other channels (clipboard, sound, etc.) still use RDP4-style packets.

An interesting side-note is that using the old protocol, you could in theory run RDP over non-TCP/IP connections (for example, you should be able to run it directly on X.25), just by rewriting small parts of the server and client. This is no longer possible with RDP5 packets, since the abstraction layers are stripped off.

3.3 Cryptographic Protocols

The communication between client and server when using RDP is protected by cryptographic protocols. The use of cryptography is however optional, so if the network administrator so desires, he can choose not to use cryptography for parts of the traffic, gaining some in performance.

The cryptography is a combination of asymmetric and symmetric cryptography, using RSA and RC4.

3.3.1 A Brief Introduction to Cryptography

In order to understand the principles behind the cryptography used in RDP, some knowledge about cryptography is needed. For a network-centric overview of cryptography please refer to [Tan02]. For a more in-depth source to cryptography information, refer to [Sch96].

Symmetric Encryption

When using symmetric encryption, both parts know a shared secret, most often known as a key, or more specific a session key. The data encrypted with the key on one end is decrypted with the same key on the other end.

The problem here is key distribution. How can the two parts agree on using a specific key in a secure way? One way is by calling the person handling the computer in the other end and agree on a key. This quickly becomes non practical when you have a need to communicate with a lot of different computers, or want to use different keys each time a new connection is setup, something that is desirable in order to limit the damage caused by a key in the wrong hands.

One of the answers to this problem is asymmetric encryption.

Asymmetric Encryption

When using asymmetric encryption, also known as public-key encryption, each key is divided into two parts. One of them is public, and the other one secret. Encryption of data is done with the public part, decryption with the private part. This means the public key is no secret, since no data can be decrypted with it. Still, it can be used to send message that only the part that owns the private part can read.

One problem with asymmetric encryption is that it is slow. When encrypting large amounts of data it is therefore better to use symmetric encryption. However, the problem with exchanging the secret key can be solved by creating a random key and then sending it to the other part using public-key encryption. This way, the best of both worlds are combined.

3.3.2 Encryption in RDP

As mentioned earlier, the encryption in RDP uses two well-known cryptographic protocols, namely RC4, a symmetric crypto for stream encryption, and RSA, an asymmetric crypto, for key setup in the session setup phase.

Session key setup

The key used for the stream encryption is negotiated during session setup as follows:

1. The server sends its public key and a random string.

2. The client encrypts another random string with the public key of the server, using the RSA algorithm.
3. The server decrypts the random string from the client with the private part of the public key it sent to the client.
4. The same RC4 session key is now created on the server and the client, based on the random strings exchanged both ways.
5. Encrypted communication using RC4 can now start.

The exact procedure for generating the session key can be found in `secure.c` of the rdesktop source code. It has not been necessary to modify this part of the software as a part of this thesis work, since the key generation procedure is the same in both version 4 and 5 of the RDP.

Asymmetric Encryption in RDP4

In RDP4, a raw RSA key is sent as a raw data object marked by a tag and a special string (*"RSA1"*). There is also a signature for the RSA key, but rdesktop does not try to verify that. Since this thesis work concentrates on RDP5 support, we will not go deeper into this subject.

Asymmetric Encryption in RDP5

When advertising to the server that we support RDP5, the server sends its public key encapsulated in a X.509 certificate structure. The reason for this is probably that Microsoft want to use standard libraries already available in Microsoft Windows for parsing the keys. X.509 is an ITU standard defined in [Int97]. There is also an Internet standard in [HPFS02] that defines the format of the certificates.

The standard is however easily misinterpreted. See [Gut00] for an in-depth reference on problems with today's different X.509 implementations. We had problems interfacing the X.509 structure sent by the server with the OpenSSL libraries [tea03], since the algorithm of the key was set to "MD5 with RSA encryption" instead of the correct "RSA encryption". This is an example of the interoperability problems of X.509. Since RDP5 is a closed protocol, this is not a problem for Microsoft since they probably use the same libraries on both client and server, which means the same software faults exist on both sides.

3.4 Security Issues

In the network world of today, a lot of information is sent using insecure networks such as the public parts of Internet. At the same time, we are becoming increasingly dependent on computers. A lot of sensitive data are carried through today's network, for example medical journals, financial data, and military information. The implications of information leaks are often very serious.

There are basically three ways of dealing with the need for privacy, integrity and authentication: plaintext over closed secure network, plaintext over public and possibly insecure networks, and encrypted data transfer over insecure public networks. We will discuss the three alternatives below.

We do see a development where previously closed networks such as Frame Relay and X.25 [Tan02] are replaced with data transfer over the Internet, protected by encryption. This change of network techniques is done because of both financial reasons–Internet connections have become much cheaper lately–and for flexibility reasons. It is much easier to find an Internet connection when you are on the road than an X.25 connection.

3.4.1 Plaintext over Closed Secure Networks

In the early days of networking, dedicated leased lines, most often Frame Relay, X.25 or similar, where the only way to go if a network connection was needed between point A and point B, for example between the offices of a company. This was often expensive and in some cases the redundancy of the connections was not that good–if a cable was damaged, the connection could be down for quite a while.

Given the fact that the networks were private, protocol security was not a top priority since the risk for information leak was low.

3.4.2 Plaintext Over Public and Possibly Insecure Public Networks

There are a large number of protocols that transfer information in plaintext over insecure public networks such as the Internet. The prime examples of this are HTTP [BLLM+99] used for the World Wide Web, and SMTP [RFC01] used for electronic mail. Both are widely used plaintext protocols used for important communication.

Sending email is sometimes compared with sending postcards–all the mailmen on the way between the sender and the receiver can read what is written on the back of the card without even the small problem of opening the envelope. The comparison is basically a good one, though the problem for a part interested in information sent using email is that the amount of emails sent daily are enormous, making the process of finding the interesting ones a major task.

There have been rumors lately (2003) about large governmental systems that monitor most email traffic in the world, searching for keywords and selecting "interesting" email and other forms of communication for manual analysis. See [Uni03] for some interesting pointers to discussions about a project named "Echelon".

There are cryptographic extensions for both HTTP and SMTP, but for the bulk of the traffic sent with both protocols, there is no guarantee that the information is neither correct nor origins from the part it claims.

3.4.3 Encrypted Plaintext Over Public Insecure Networks

As mentioned earlier, we see a development where more and more information is sent over the public networks, due to financial and flexibility (practical) reasons. In the cases where leased lines are replaced by Internet connections, most often some kind of protection mechanism is added.

One of the available methods for protecting the data is to use a Virtual Private Network (VPN) [Tan02], a technique where all traffic between two sites are protected using a so-called tunnel that is encrypting all the traffic before sending it over the insecure network. The advantage of this method is that no changes are needed in the applications used, they can still use plaintext communication. Among the disadvantages are the fact that it is very often hard to setup VPNs in a correct way. Also, the price of the software and hardware needed tend to be quite high. Another disadvantage is that clients connecting via VPN to your internal network need to be as secure as your internal clients, or security problems on the VPN client will give an attacker access to the whole network.

Another method is to make sure that applications transferring sensitive information encrypt their traffic. This way, all non-sensitive data must not be encrypted, which decreases the amount of CPU power needed on both sides of the connection.

See section 3.3 for more discussion about cryptographic protocols.

3.4.4 Do Secure Networks Really Exist?

A relevant question here is whether there is such a thing as a secure network. Some people tend to believe that the network inside their firewall is secure, so there is no need to update the computers on the network and all traffic can safely be plaintext.

The danger with this assumption is that in order for it to be correct, all parts on the internal network must be trusted. It is a fact that a lot of attacks originate from for example employees who are unhappy with their employer for some reason. Also, if one single computer on the internal network is compromised, it can be used as a gateway to attack all the poorly protected computers on the internal network.

Using plaintext protocols for sensitive data on internal networks is also a bad idea since there exists methods for gaining access to the network path between two computers given access to the same broadcast domain as one of them. Such a method is ARP-spoofing, covered in [dVdVI98].

During our investigations of RDP we found that none of the available RDP clients from Microsoft verified the public key of the server. This is a rather serious security flaw as it opens up for MITM attacks such as the one we used to find information about the protocol sent over the network (see section 2.3.1). We have documented this problem and how we reported it in appendix B. The security problem was also covered by a Swedish newspaper, see [Nor04].

Chapter 4

Implementation

In this chapter we discuss software maintenance in general, how rdesktop works and how we introduced support for RDP5 in the software.

4.1 Understanding Existing Software

Part of the challenge in this final thesis is to understand and modify an existing piece of software, rdesktop [rt03]. Some of this activity is traditional software maintenance, an area where there is academic research. In this section, we will introduce the reader to the subject.

4.1.1 About Software Maintenance in General

A standard definition adopted by the IEEE in 1983 of "Software maintenance" follows: *Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment* [Ins83]. This is a very broad definition. In [LPW88], four main forms of software maintenance are listed:

1. **Corrective Maintenance:** eliminating errors in the program functionality.

2. **Adaptive Maintenance:** modifying the application to meet new operational circumstances (such as a new environment).
3. **Perfective Maintenance:** enhancement (new operations and refinements to old functions).
4. **Preventative Maintenance:** modifying a program to improve its future maintainability.

The implementation part of this thesis work is mostly about the third form, perfective maintenance, since we add new functionality (the support for version 5 of the Remote Desktop Protocol). However, it is also adaptive maintenance, since we are modifying the program to enable its functionality in a new environment–a new version of the Windows Terminal Services Protocol. According to [LH93], about 80% of all software maintenance is either perfective or adaptive. Of the remaining 20%, almost all is put into corrective maintenance.

4.1.2 Tools for Understanding Existing Software

We have found that the number of good tools to help in software maintenance activities are rather low. However, since rdesktop is a small program (about 10,000 lines of code in C), rather simple tools are needed. Among the tools used in the understanding of rdesktop are:

- **cflow** is a program that, given C-code, prints out the flow of the program in a textual form. Each function call is listed together with a reference to where the function called is located in the source analyzed. This is most useful in the beginning of the analysis, since it gives a quick overview of the program. After a while, you tend to remember the relationships between different functions.
- **Emacs** is, according to its webpage [Fou04a], "the extensible, customizable, self-documenting real-time display editor". It has many features that help when developing software. Among them are:
  - **etags** is a utility program that produces a "TAGS" file. Emacs reads the file produced and provides a function that makes it

  easy to jump to the definition of a called function, a global variable and other symbols you need to find the definition of when following the flow of a program.

  – **speedbar** provides a window where you can navigate functions defined in different source files in the directory you are currently editing.

- **grep** is the standard Unix utility for finding text, for example function definitions, in files. It is possible to call grep from within emacs. That way you get the results in a list Emacs can parse, making it possible to jump to a certain search result using a keystroke.

4.2 Structure of the Software

In order to make further maintenance activities on rdesktop easier, we intend to give a description of how rdesktop works internally.

4.2.1 A Well Designed Layered Network Program

As we have seen in section 3.1, RDP is a layered protocol. Rdesktop implements this using a very naturally layered design where each layer in the network stack is also implemented as a layer in the software. Each layer is implemented in its own file for easy navigation. See figure 4.1.

4.2.2 The Files in the Rdesktop Source code

As a reference for future developers we will describe the different files in the rdesktop source code with a small comment about each of them.

- `rdesktop.c` contains the *main* routine, command line parsing, and support routines used by the rest of the program.

- `xwin.c` contains the code for the graphic interface connecting to a X Server. There are alternative graphic interfaces such as a SVGA driver, although we will not cover them here since they are not part of the standard rdesktop distribution.

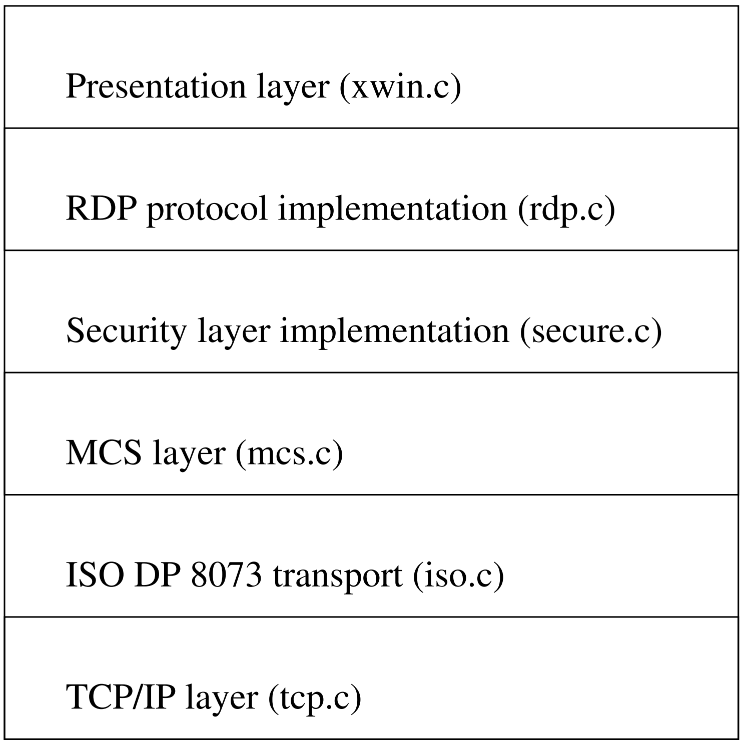
Presentation layer (xwin.c)
RDP protocol implementation (rdp.c)
Security layer implementation (secure.c)
MCS layer (mcs.c)
ISO DP 8073 transport (iso.c)
TCP/IP layer (tcp.c)

Figure 4.1: Structure of rdesktop

- `rdp.c` contains code for sending and receiving RDP packets. A lot of this code is used only during protocol setup.
- `rdp5.c` contains code that takes care of RDP5 packets.
- `mcs.c` contains code that parses and sends MCS packets.
- `iso.c` contains code for parsing and sending ISO DP 8073 packets.
- `tcp.c` contains code for sending and receiving tcp packets.
- `ewmhints.c` contains routines for communication with the window manager.
- `licence.c` contains routines for license handling. See section B.4 for discussion about how the licenses work.

- `orders.c` contains code that processes orders (graphic data) from the server.
- `parse.h` contains macros that operate on a STREAM structure (also defined in this file). The macros are used to read and write data from and to the server in a endian-independent way.
- `proto.h` contains prototypes for the public functions in the different source files.
- `types.h` contains most of the types used in rdesktop.
- `scancodes.h` contains keyboard related definitions.
- `secure.c` contains cryptography-related code such as parsing of public keys and encryption/decryption of data.
- `xkeymap.c` is used for keyboard mapping in the X Window System.
- `bitmap.c` contains routines for decompressing bitmaps (such as icons and other small pictures) sent from the server.
- `cache.c` contains cache routines for bitmaps, fonts, desktops, and pointers. Bitmaps, fonts, desktops, and pointers are sent only once and then referenced to as cache identifiers in the protocol. A related note is that fonts are not sent as a whole, but only the characters needed right now are sent and cached. The server then keeps track of which characters it has sent already and what cache identifiers are on the client side.
- `channels.c` contains code for registering callback functions that should be called whenever there are data on one of the RDP5 channels.
- `cliprdr.c` contains code that handles clipboard data between X Window System and Windows, and between different rdesktop sessions. See section 4.4 for a discussion about this.
- `ipc.c` contains code for communication between different rdesktop instances connected to the same X server. This is used by the clipboard code. The communication is performed using X properties on the root window, using code in xwin.c.

The last three files (`channels.c`, `cliprdr.c` and `ipc.c`) are completely new and an outcome of this final thesis.

4.2.3 Program Structure

We will here give a small introduction to how rdesktop works internally for future developers and our own reference.

When rdesktop starts, the *main* method in `rdesktop.c` is executed. It parses the command line options (using the *getopt* function, standard for most Unix programs), initializes the graphic user interface, sets up some other things (interprocess communication, clipboard) and then tries to connect to the server specified on the command line.

The connection is made by calling *rdp_connect*, defined in `rdp.c`. This function calls *sec_connect*, defined in `secure.c` which initializes some data and calls *mcs_connect*, defined in `mcs.c`. MCS first opens an ISO connection using *iso_connect* from `iso.c` and then negotiates its connection parameters, including the channels needed for transferring graphics, clipboard, sound and other data.

The ISO layer in `iso.c` creates it is own connection by using routines from `tcp.c`.

After connection setup, the *main* function in `rdesktop.c` opens the window (using UI functions from `xwin.c`) and then calls *rdp_main_loop*, defined in `rdp.c`. This is the main program loop.

The main program loop works by trying to read packets from the server, handling them as they are received. Reading a packet is a multi layer process, where *rdp_recv* calls *sec_recv* which calls *mcs_recv*, which calls *iso_recv* which in turn calls *tcp_recv*. Worth noting here is that the data structure received is statically allocated in `tcp.c`, although the actual data size is allocated dynamically each time a packet is received using *xrealloc*, defined in `rdesktop.c`.

All packet handling is done on *STREAM* structures using macros from `parse.h`.

Sending packets is a similar process. When sending an RDP packet, you initialize a *STREAM* structure using the function *rdp_init_data* and giving a maximum length of the packet. This function calls *sec_init* with flags and a max length, adding the length of the RDP header to the max length it got as an argument. This process is repeated down through the MCS, ISO and TCP layers. After initializing the packet, the data is filled into the *STREAM* structure using macros from `parse.h`.

4.3 The Implementation of RDP5 in rdesktop

The implementation of RDP5 in rdesktop was done in two stages. First, we made sure we could speak RDP5 with the server with the same feature set as with RDP4. Then we added support for channels which opened the path for clipboard support. The implementation of clipboard support is covered in section 4.4.

One of the major goals of the implementation was to keep as much of the original code as untouched as possible, and to provide an option to run with RDP4 if desired. Therefore, we decided to implement a command line flag for RDP5 support. In the current version (autumn 2003), RDP4 is still the default and RDP5 can be enabled using a flag. Rdesktop v2.0 will probably use RDP5 by default.

All the RDP5-specific code is within conditional blocks that trigger on the value of a variable named *use_rdp5*. However, even when running in RDP4 mode, some of the output on the network has changed compared to previous version. This is, however, normal–the native RDP5 clients output this data as well.

We list the changes needed for RDP5 support:

- `Makefile`: Added rdp5.o to the list of objects to be compiled.
- `configure`: Added support for a new debug primitive.
- `constants.h`: Added constants used in RDP5.
- `iso.c` Made connection request contain user name. Made packet receiving code recognize RDP5 packets, forwarding them not to the MCS layer but to the RDP5 handling code in `rdp5.c`.
- `bitmap.c`: Added support for new bitmap compression (does not use a useless length field in the T.128 specification anymore).

- `mcs.c`: Added new target parameters (more channels). This provides better length calculation for the outgoing MCS data. Also, new functions for sending data to a specific MCS channel were added.
- `orders.c`: Made sure not to discard bufsize, row_size and final_size when handling BMPCACHE requests, they are used in RDP5.
- `rdesktop.c`: Added new command line flags and their parsing.
- `rdesktop.h`: Added new debug primitives and rdp5 channel structure definitions.
- `rdp.c`: Added new logon packet. Trigger server to start sending RDP5 packets by setting a value in the general capability set.
- `secure.c`: Added code for parsing the new type of cryptography-related packets of RDP5-type (X.509 based). Also added code that tells the server we support RDP5.
- `rdp5.c`: Implemented parsing of RDP5 packets and dispatching them to the correct order or cache handling routine.

4.4 Implementation of Clipboard Support

Implementing clipboard support in rdesktop was a surprisingly hard task. The actual implementation was done in a few days, but we had to do a lot of thinking before finding out all the principles.

The reason this was such a hard task is that the principles behind the clipboards in the X Window System and Microsoft Windows are very different and that the clipboard in X Window System is not very well documented.

4.4.1 Introduction to the Clipboard in the X Window System

The clipboard in the X Window System is documented in [RM93], section 2. This text is, however, written for people already familiar with the concepts of the X Window System. Another text is [Ben93], which describes the concepts involved in a better way. However, the examples in the latter uses the Xt toolkit, something we do not want to do in rdesktop (it cannot be done without rewriting large parts of the user interface code, and we do not want to depend on the Xt library).

The basic principle is that a window where a user makes some kind of gesture that indicates the user wants to select something acquires ownership of a so called selection. When another window detects a gesture indicating that the user of it wants to paste, it checks whether any application owns the selection. If so, it asks the application for what clipboard formats are available, and then asks for the data itself using a format code. The format codes are in form of so called atoms (an X Window System term, see [GS96]). The available formats are defined both in the ICCCM ([RM93]) and by applications themselves. That is, two applications of the same type can exchange information in their own format if desirable.

We call this a pull system–it is the client that wants the data that must check what formats are available, if any.

There are two selections, PRIMARY and CLIPBOARD. However, how they should be used is not regulated in a formal standard but instead relies on common understanding in the X Window System programming world. [Ben93] proposes some ways of solving this, and we try to follow this in our implementation.

4.4.2 Introduction to the Clipboard in Microsoft Windows

The clipboard in Microsoft Windows is documented in [Cor03]. Basically, when an application puts something in the clipboard, all other applications get a notification. This way they can lighten up their paste buttons/menu options. Embedded in the notification is a list of formats available, meaning a text application can choose not to enable pasting when it knows it cannot handle any of the formats available.

We call this a push system–the client with the data tells the other clients it has data.

4.4.3 Clipboard and RDP

By studying the data stream from the server when executing different clipboard operations on a native Windows client (see section 2.3.2), we discovered several facts about the type of protocol used for sending clipboard data over an RDP5 connection. A virtual channel in the MCS layer is used to transfer the data packets. Just as with the clipboard when running locally on Windows, a clipboard announcement is sent when there is clipboard data available on any of the two sides of the connection. For details about the clipboard packets, refer to section A.3.2.

4.4.4 Merging the Principles of the X Window System and Microsoft Windows

There are several subproblems in implementing clipboard support in rdesktop. The underlying problem is to merge the push system in Microsoft Windows, as discussed in section 4.4.2, with the pull system in the X Window System, discussed in section 4.4.1. Apart from that, there are other problems:

1. In order to support clipboard transfers from X applications to Windows, we need to know when there are clipboard data available. There is no natural way to do that.
2. We need to map X Window System formats to Microsoft Windows formats.
3. There is sometimes a need for conversion between different formats. Text can sometimes be received in UTF-8 in X, but the Unicode text format transported to Windows seems to be UCS-2.

We chose to handle the first problem with a simple approach. Each time we start rdesktop, we announce a static list of formats available to the Windows server. This means we do not care if there is any clipboard data available in the X Window System. Also, each time we have transferred data to Windows, we send another announcement in order to invalidate the server-side clipboard cache. If we do not do this, Windows will not ask for clipboard data the next time it wants to paste, and since there is no way to know when the clipboard in X changes owner, that would mean we do not always get the correct data.

When there is a request for data from the Microsoft Windows server, we check if there is any data available on PRIMARY and then on CLIPBOARD. If there is, we send this. Otherwise, we just send an empty string, something we must do in order to avoid hung applications on the server side.

Regarding the second problem, we choose to implement a rather limited version of clipboard support. We only support transfer of plain text between Windows and X (both directions). This means we can limit the amount of conversion needed. Other formats will probably be implemented in the future, but that is outside the scope of this final thesis. Also, we do not support any conversion of UTF8 to UCS-2 (two different ways of representing character data) or vice versa. We do however support transfer of all Windows clipboard formats between two rdesktops. That is, if the user cuts or copies from a Windows application in one rdesktop, pasting in another rdesktop running another Windows application will work seamlessly, retaining whatever format the Windows applications find appropriate. This is implemented using a simple interprocess communication between the different rdesktop processes.

Chapter 5

Discussion and Conclusion

In this chapter, we will discuss our work and summarize what has been done. We will also list some possible future work.

5.1 Discussion

When we began working on this thesis work, we had a quite well-defined problem to solve– the need for clipboard support in Rdesktop. This problem proved to contain a large set of subproblems, many of them impossible to anticipate in the beginning.

The subproblems ranged from large questions such as what method to use for the reverse-engineering needed, to small questions, such as what datatype was hidden in a small part of the datastream.

During our work, we simply tried to solve the problems as they occured. We identified the problem, found a set of possible solutions, and chose the solution that fit best into the work we had already done. Since our workflow was an iterative process (see section 2.3.2), we were prepared for that we might have to reimplement some things several times in order to get the desired results.

We found the task interesting and stimulating. Even though it took a while to gain results that are possible to explain to non-experts on the subject, we could see progress almost every week during the period when we were investigating the protocol and implementing support for it in rdesktop.

5.2 Conclusion

In section 1.2, we motivated this thesis work by the following reasons:

1. *Enhance rdesktop in order to support RDP5 and clipboard*
2. *Provide documentation in order to ease further development of rdesktop*

Regarding the first item, we have succeeded in adding support for RDP5 to rdesktop, including generic code for virtual channel handling. We have also succeeded in adding basic support for clipboard operations. About 2000 lines of code in use today in rdesktop were created as a result of this final thesis.

Regarding the second item, we believe new and current developers of rdesktop will find this report helpful as a reference.

Apart from fulfilling the goals motivating the thesis, we have developed a method, the iterative process described in section 2.3.2, for our Reverse Engineering needs. We have also developed and enhanced several tools for our work:

1. *rdpproxy* was enhanced to be able to correctly read and forward RDP5 messages. By using an existing tool, we definitely saved large amounts of time.
2. *pparser.py* was developed from scratch to ease parsing and understanding output from rdpproxy. Developing a tool for this purpose was a great timesaver. Without this tool, we most probably would not have succeeded in fulfilling the goals of the thesis work.

   The parser also will be of use in further rdesktop development, and the structure of it might be of interest for other people in need of writing packet parsers, since we developed an object-oriented way of parsing packets that proved to work very well.

3. *Our diary* that we mentioned in 2.4 proved very useful, making it possible to go back to information found earlier.

5.3 Future work

As of autumn 2003, there are a number of things to be done on rdesktop to support all of RDP5. Among them are:

1. Sound redirection (virtual channel).
2. Serial and parallel port redirection (virtual channel).
3. Disk drive redirection (virtual channel).
4. Capability negotiation (allow/disallow for example desktop backgrounds and themes). The packet handling for this information is hardcoded in rdesktop at the time of writing.
5. Support for more clipboard formats when transferring data between the X Window System and Microsoft Windows.
6. Support for compression of the data packets.

Also, the security layer should be able to verify the peer when connecting in order to prevent man in the middle attacks, and it should also verify the signature of data packets. This functionality is missing at the time of writing.

Another area of interest is to develop a software that can connect to servers running Citrix Metaframe [Inc04]. It might be possible to use Rdesktop for this purpose since the protocols are related.

5.4 Concluding Remarks

During the work, we found and reported some rather serious security flaws in Microsoft's client implementations. We have learned and discussed the theory behind the flaws, and some of the ethics behind reporting such flaws in a responsible manner. See appendix B.

We have learned a lot about network protocols in general, how to parse binary protocols, and how to do reverse-engineering of them. This is knowledge that will be of great use for us in the future.

Worth noticing is that our implementation of RDP5 and clipboard support has been made entirely without use of commercial software. All software used in the process was either open source [ini03] or developed inhouse. The only commercial software involved was the Windows servers used to test the functionality.

Bibliography

[Ben93] Chan Benson. *Implementing Cut and Paste in the X Environment*. Hewlett Packard Company, first public release edition, October 1993.

[BLLM+99] T. Berners-Lee, P. Leach, L. Masinter, H. Frystyk, J. Mogul, J. Gettys, and R. Fielding. *Hypertext Transfer Protocol - HTTP/1.1, RFC2616*. UC Irvine, Compaq, W3C, Xerox, Microsoft, MIT, June 1999.

[Com04] Gerald Combs. Ethereal: A network protocol analyzer. Website, Accessed June 2004. http://www.ethereal.com/.

[Cor03] Microsoft Corporation. Clipboard. Website, Accessed January 2003. http://msdn.microsoft.com/library/.

[cotEc] The council of the European communities.

[CP98] Crispin Cowan and Calton Pu. Death, taxes, and imperfect software: surviving the inevitable. In *Proceedings of the 1998 workshop on New security paradigms*, pages 54–70. ACM Press, 1998.

[dev03] The Samba developers. Samba, server and client for the microsoft smb (cifs) protocol. Website, Accessed April 2003. http://www.samba.org/.

[dVdVI98] Marco de Vivo, Gabriela O. de Vivo, and Germinal Isern. Internet security attacks at the basic levels. *ACM SIGOPS Operating Systems Review*, 32(2):4–15, 1998.

[Ent04] HOB Enterprise. Hoblink jwt - java client for windows terminal servers. Website, Accessed November 2004. http://www.hob.de/www s/produkte/connect/jwt.htm.

[FC04] Erik Forsberg and Matt Chapman. rdp-proxy. Website, Accessed November 2004. http://cvs.sourceforge.net/viewcvs.py/rdesktop/rdpproxy/.

[Fou03] The Python Software Foundation. Python language website. Website, Accessed May 2003. http://www.python.org/.

[Fou04a] Free Software Foundation. Emacs - extensible, real-time editor. Website, Accessed June 2004. http://www.gnu.org/directory/emacs.html.

[Fou04b] X.org Foundation. The x window system. Website, Accessed June 2004. http://www.x.org/.

[GS96] James Gettys and Robert W. Scheifler. *Xlib - C Language X Interface.* Cambrigde Research Laboratory, DEC, MIT, x11r6.4 edition, 1996.

[Gut00] Peter Gutman. X.509 style guide. Website, Accessed October 2000. http://www.cs.auckland.ac.nz/ pgut001/pubs/x509guide.txt.

[HPFS02] R. Housley, W. Polk, W. Ford, and D. Solo. *Internet X.509 Public Key Infrastructure, RFC3280.* RSA, NIST, VeriSign, Citigroup, April 2002.

[Inc04] Citrix Sytems Inc. Citrix metaframe access suite. Website, Accessed June 2004. http://www.citrix.com/.

[ini03] The Open Source initiative. The open source initiative website. Website, Accessed May 2003. http://www.opensource.org/.

[Ins83] Institute of Electrical and Electronic Engineers. *IEEE Standard Glossary of Software Engineering Terminology*, 1983.

[Int97] International Telecommunications Union. *ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*, June 1997.

[Lab04] Lawrence Berkeley National Laboratory. tcpdump. Website, Accessed June 2004. http://www.tcpdump.org/.

[LH93] Kevin Lano and Howard Haughton. *Reverse Engineering and software maintenance : A practical approach.* McGraw Hill, 1993.

[LPW88] S. Leonard, J. Pardoe, and S. Wade. Software maintenance - cinderella is still not getting to the ball. In *BSC/IEE Conference on Software Engineering*, pages 104–106, 1988.

[Mic] Sun Microsystems. snoop. Software. Included in the Solaris operating system.

[Mic04a] Microsoft. Microsoft knowledge base article - 186607, understanding the remote desktop protocol (rdp). Website, Accessed June 2004. http://support.microsoft.com/.

[Mic04b] Microsoft. Microsoft word. Software, Accessed June 2004. Part of Microsoft Office, http://office.microsoft.com/.

[Mic04c] Microsoft. Remote desktop connection client 1.0.3 for mac os x. Website, Accessed November 2004. http://www.microsoft.com/mac/downloads.aspx.

[Mic04d] Microsoft. Remote desktop connection client software download. Website, Accessed November 2004. http://www.microsoft.com/windowsxp/downloads/tools/rdclientdl.mspx.

[Mic04e] Microsoft. Windows server 2003 terminal services. Website, Accessed June 2004. http://www.microsoft.com/terminalservices/.

[Mic04f] Sun Microsystems. Staroffice 7 office suite. Website, Accessed June 2004. http://wwws.sun.com/software/star/staroffice/.

[MJS$^+$00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 47–60. ACM Press, 2000.

[MPP97] Microsoft, PictureTel, and Polycom. *Application Sharing.* International Telecommunications Union, March 1997.

[Nor04] Anders Nordner. Svenskar hittar hål i windows. *Computer Sweden*, April 2004.

[Pre00] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx, and tcl for a search/string-processing program. Technical report, Universität Karlsruhe, Fakultät für Informatik, Germany, March 2000.

[Pro05] The LaTeX Project. Latex - a document preparation system. Website, Accessed November 2005. http://www.latex-project.org/.

[PY97] Y. Pouffary and A. Young. *ISO Transport Service on top of TCP (ITOT), RFC2126.* Digital Equipment Corporation, ISODE Consortium, March 1997.

[RFC01] AT&T Laboratories. *Simple Mail Transfer Protocol, RFC2821*, April 2001.

[RM93] David Rosenthal and Stuart W. Marks. *Inter-Client Communication Conventions Manual.* Sun Microsystems, Inc., 2.0 edition, December 1993.

[rt03] The rdesktop team. Rdesktop, a client for the microsoft remote desktop protocol. Website, Accessed April 2003. http://rdesktop.sf.net/.

[Sch96] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., second edition, 1996.

[Sec03a] Securityfocus. Bugtraq. Website, Accessed May 2003. http://www.securityfocus.org/.

[Sec03b] Securityfocus. Bugtraq frequently asked questions. Website, Accessed May 2003. http://www.securityfocus.org/popups/forums/bugtraq/faq.shtml.

[STD81] Defense Advanced Research Projects Agency. *Transmission Control Protocol, STD007*, September 1981.

[T.193a] International Telecommunications Union. *Generic Conference Control Protocol Specification*, November 1993.

[T.193b] International Telecommunications Union. *Multipoint Communication Service Protocol Specification*, November 1993.

[Tan02] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, fourth edition, 2002.

[tea03] The OpenSSL team. A open source implementation of the secure sockets layer. Website, Accessed March 2003. http://www.openssl.org/.

[Tri03] Andrew Tridge. Response on the samba-technical mailinglist to a question about formal reverse-engineering methods. Mailing list, Accessed January 2003. http://lists.samba.org/pipermail/samba-technical/2003-January/041701.html.

[Uni03] American Civil Liberties Union. Echelon watch. Website, Accessed April 2003. http://www.echelonwatch.org/.

[Wik05a] Wikipedia. Basic encoding rules. Website, Accessed November 2005. http://en.wikipedia.org/wiki/Basic ncoding ules.

[Wik05b] Wikipedia. Disassembler. Website, Accessed November 2005. http://en.wikipedia.org/wiki/Disassembler.

[Wik05c] Wikipedia. Man in the middle attack. Website, Accessed November 2005. http://en.wikipedia.org/wiki/MITM.

Appendix A

A complete RDP session, packet by packet

In this appendix, we list the different kinds of packets involved in RDP5. We also give some comments related to the contents of them in order to clarify things we have found hard to understand.

The motivation behind this appendix is that we want to ease further development of rdesktop by providing better documentation.

This appendix contains a number of tables representing parts of network packets transferred as part of RDP. The tables were generated using `pparser.py` and the following command line options: `./pparser.py -f LATEX -n -l  /xjobb/report/figures/pktfigs/ ../sniff/report.1.out`.

An example table is table A.1.

C2		TPKT from Client		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x019c (412)
4	TPDU			See **C2-0**

Table A.1: Example packetpart table

Let us examine the different parts of the table.

At the top left corner of the table is a packetpart identifier. The identifier begins with either an **S**, meaning the packet originated at the server side of the connection, or a **C**, meaning the packet originated on the client side of the connection.

Following the character is a number which is the sequence number for the whole packet. There is one sequence number for the server, and one for the client. That is, C1 comes before C2, but S1 might have arrived from the server in between. The text around the figures will explain the relationships between the packets.

Each network packet is represented by 0 to n tables in order to get tables that are small enough to fit into the pages of the document, yet still readable. The identifier of a subpart have a dash (**-**) after the character and packet sequence number, and then another sequence number identifying the subpacket. A subpacket can have another subpacket, identified by another dash and a new sequence number. As an example, **S1-0-0** is a subpacket of **S1-0** which is a subpacket of **S1**.

Please note that a subpacket in this context does not necessarily mean it is a subpacket protocolwise. The subpackets here are often subpackets because of the size of the table which had to fit into one page for clarity.

Continuing with the upper right corner, there is a textual description of the packetpart.

The next line of the table is the table header. We will explain each column below:

- **Offset** is the offset within the packet for this particular data. The offset begins at 0 and is measured in octets (which is the same as a byte on most machines).

- **Datatype** is the datatype of the data. There are two main types of datatypes. One is “primitive” datatypes like integers, strings, and raw bytedata. The other one is pointers to subpacket. In the latter case, the Description and Expected Value columns are empty, and the Value column is a reference to a subpacket.

- **Description** is a textual description of the data. In some cases this is empty, which most often means we do not know the function of the data.

- **Exp. val** is the expected value of the data. During our reverse-engineering process this was of great help in finding when a packet was abnormal, or when our parsing went wrong. In some cases it has information about the remaining length of a packet calculated py `pparser.py` to be compared with the remaining length value according to the packet.

- **Value** is the real value of the data. It can also be a pointer to a subpacket table.

Following the table header is the actual data in the packetpart, one line per value.

A.1 Protocol Phases

The protocol has two basic phases. The setup phase, where a TCP connection is made, channels are allocated, and a login packet is sent (although login can also be done manually): and the session phase where normal operations occur, like graphic data being sent from the server, and mouse and keyboard operations are being sent from the client.

We will describe the different types of packets exchanged during the different phases below.

A.2 The Setup Phase

A.2.1 ISO DP 8073 Connection Setup

The setup phase starts from the bottom of the protocol graph (figure 3.1) by setting up an ISO DP 8073 connection between the parts.

C1 TPKT from Client

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x0023 (35)
4	TPDU			See **C1-0**
17	Latin1 String(22)	mstshash		Cookie: mstshash=demo1
39	Int16 (le)	Unknown		0x0a0d (2573)

C1-0 TPDU

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPDU hdr length		0x1e (30)
1	Int8 (be)	TPDU packet type		0xe0 (224) Connection request
2	Int16 (be)	Dst ref		0x0000 (0)
4	Int16 (be)	Src ref		0x0000 (0)
6	Int8 (be)	Class		0x00 (0)

Table A.2: ISO Connection Request

The ISO Connection Request, illustrated in table A.2, is the first packet sent from the client when initiating an RDP connection. This packet is part of ISO DP 8073 defined in RFC2126 [PY97]. The packet is a TPDU (defined in ISO 8073) encapsulated in a so called TPKT (defined in RFC2126).

Note that the mstshash and the 16 bit integer in the TPKT is not part of the standard, but a Microsoft extension. We do not know the function of this data, but it could have something to do with the Active X control version of the client and its integration with a web server.

S1		TPKT from Server		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x000b (11)
4	TPDU			See **S1-0**

S1-0		TPDU		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPDU hdr length		0x06 (6)
1	Int8 (be)	TPDU packet type		0xd0 (208) Connection confirm
2	Int16 (be)	Dst ref		0x0000 (0)
4	Int16 (be)	Src ref		0x1234 (4660)
6	Int8 (be)	Class		0x00 (0)

Table A.3: ISO Connection Response

As a response to the connection request, the server sends a Connection Response, also part of the ISO DP 8073. This packet is illustrated in table A.3.

A.2.2 Multipoint Communication Service Protocol Setup, Connect Initial

Now when the client has a transport for its MCS packets, it starts setting up the next layer in the protocol graph, the Multipoint Communication Service protocol (MCS), defined in [T.193b]. The MCS setup packet (named MCS Connect Initial in the documentation) is packed with a lot of data. Due to the large amounts of data, we have had to split the packet into different tables. See tables A.4 to A.14 for the different parts.

C2 TPKT from Client

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x019c (412)
4	TPDU			See **C2-0**

C2-0 TPDU

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	TPDU hdr length		0x02 (2)
1	Int8 (be)	TPDU packet type		0xf0 (240) Data
2	Int8 (be)	TPDU eot	128	0x80 (128)
3	MCS packet			See **C2-0-0**

C2-0-0 MCS packet

Offset	Datatype	Description	Exp. val.	Value
0	MCS Connect Initial			See **C2-0-0-0**

Table A.4: MCS Connect Initial, TPKT, ISO, and MCS packet headers.

First in the packet are the TPKT and TPDU packet headers, see table A.4. The packetpart **C2-0-0** is mainly there because of a design problem in `pparser.py`, but due to time constraints we did not fix that.

C2-0-0-0		MCS Connect Initial		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	BER tag 32613	MCS Conn initial	32613	32613
2	VariableInt(2)	BER length		0x0190 (400)
5	BER tag 4	OCTET-_STRING	4	4
6	VariableInt(1)	BER length		0x0001 (1)
7	KLIInt(8)	Calling Domain Value		0x01 (1)
8	BER tag 4	OCTET-_STRING	4	4
9	VariableInt(1)	BER length		0x0001 (1)
10	KLIInt(8)	Called Domain Value		0x01 (1)
11	BER tag 1	BOOLEAN	1	1
12	VariableInt(1)	BER length		0x0001 (1)
13	KLIInt(8)	Upward flag Value		0xff (255)
14	Domain Parameters	Target		See **C2-0-0-0-0**
33	Domain Parameters	Min		See **C2-0-0-0-1**
52	Domain Parameters	Max		See **C2-0-0-0-2**
71	MCS initial userdata			See **C2-0-0-0-3**
90	[unknown type]	Remaining MCS conn initial data		See **C2-0-0-0-4**

Table A.5: MCS Connect Initial

BER is an abbreviation for Basic Encoding Rules [Wik05a] , a way of packing integers for network transfer.

The more interesting contents of the packet starts in table A.5. This packet is defined in section 10.1 in [T.193b]. It consists of several parts, the largest being the three domain parameter sets in table A.6, A.7 and A.8. These define for example how many channels the clients might open and other parameters. The fourth large part is the User data block, visualized in table A.9.

C2-0-0-0-0		Domain Parameters Target		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	BER tag 48	Target	48	48
1	VariableInt(1)	BER length		0x0019 (25)
2	BER tag 2	INTEGER	2	2
3	VariableInt(1)	BER length		0x0001 (1)
4	KLIInt(8)	Channels Value		0x22 (34)
5	BER tag 2	INTEGER	2	2
6	VariableInt(1)	BER length		0x0001 (1)
7	KLIInt(8)	Users Value		0x02 (2)
8	BER tag 2	INTEGER	2	2
9	VariableInt(1)	BER length		0x0001 (1)
10	KLIInt(8)	Tokens Value		0x00 (0)
11	BER tag 2	INTEGER	2	2
12	VariableInt(1)	BER length		0x0001 (1)
13	KLIInt(8)	Priorities Value		0x01 (1)
14	BER tag 2	INTEGER	2	2
15	VariableInt(1)	BER length		0x0001 (1)
16	KLIInt(8)	Throughput Value		0x00 (0)
17	BER tag 2	INTEGER	2	2
18	VariableInt(1)	BER length		0x0001 (1)
19	KLIInt(8)	Height Value		0x01 (1)
20	BER tag 2	INTEGER	2	2
21	VariableInt(1)	BER length		0x0002 (2)
22	KLIInt(16)	PDUsize Value		0xffff (65535)
24	BER tag 2	INTEGER	2	2
25	VariableInt(1)	BER length		0x0001 (1)
26	KLIInt(8)	Protocol Value		0x02 (2)

Table A.6: MCS Connect Initial, Domain Parameters (Target)

C2-0-0-0-1		Domain Parameters Min		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	BER tag 48	Min	48	48
1	VariableInt(1)	BER length		0x0019 (25)
2	BER tag 2	INTEGER	2	2
3	VariableInt(1)	BER length		0x0001 (1)
4	KLIInt(8)	Channels Value		0x01 (1)
5	BER tag 2	INTEGER	2	2
6	VariableInt(1)	BER length		0x0001 (1)
7	KLIInt(8)	Users Value		0x01 (1)
8	BER tag 2	INTEGER	2	2
9	VariableInt(1)	BER length		0x0001 (1)
10	KLIInt(8)	Tokens Value		0x01 (1)
11	BER tag 2	INTEGER	2	2
12	VariableInt(1)	BER length		0x0001 (1)
13	KLIInt(8)	Priorities Value		0x01 (1)
14	BER tag 2	INTEGER	2	2
15	VariableInt(1)	BER length		0x0001 (1)
16	KLIInt(8)	Throughput Value		0x00 (0)
17	BER tag 2	INTEGER	2	2
18	VariableInt(1)	BER length		0x0001 (1)
19	KLIInt(8)	Height Value		0x01 (1)
20	BER tag 2	INTEGER	2	2
21	VariableInt(1)	BER length		0x0002 (2)
22	KLIInt(16)	PDUsize Value		0x420 (1056)
24	BER tag 2	INTEGER	2	2
25	VariableInt(1)	BER length		0x0001 (1)
26	KLIInt(8)	Protocol Value		0x02 (2)

Table A.7: MCS Connect Initial, Domain Parameters (Minimum)

C2-0-0-0-2		Domain Parameters Max		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	BER tag 48	Max	48	48
1	VariableInt(1)	BER length		0x001c (28)
2	BER tag 2	INTEGER	2	2
3	VariableInt(1)	BER length		0x0002 (2)
4	KLIInt(16)	Channels Value		0xffff (65535)
6	BER tag 2	INTEGER	2	2
7	VariableInt(1)	BER length		0x0002 (2)
8	KLIInt(16)	Users Value		0xfc17 (64535)
10	BER tag 2	INTEGER	2	2
11	VariableInt(1)	BER length		0x0002 (2)
12	KLIInt(16)	Tokens Value		0xffff (65535)
14	BER tag 2	INTEGER	2	2
15	VariableInt(1)	BER length		0x0001 (1)
16	KLIInt(8)	Priorities Value		0x01 (1)
17	BER tag 2	INTEGER	2	2
18	VariableInt(1)	BER length		0x0001 (1)
19	KLIInt(8)	Throughput Value		0x00 (0)
20	BER tag 2	INTEGER	2	2
21	VariableInt(1)	BER length		0x0001 (1)
22	KLIInt(8)	Height Value		0x01 (1)
23	BER tag 2	INTEGER	2	2
24	VariableInt(1)	BER length		0x0002 (2)
25	KLIInt(16)	PDUsize Value		0xffff (65535)
27	BER tag 2	INTEGER	2	2
28	VariableInt(1)	BER length		0x0001 (1)
29	KLIInt(8)	Protocol Value		0x02 (2)

Table A.8: MCS Connect Initial, Domain Parameters (Maximum)

C2-0-0-0-3		MCS initial userdata		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	BER tag 4	Userdata length	4	4
1	VariableInt(2)	BER length		0x012f (303)
4	Int16 (be)		5	0x0005 (5)
6	Int16 (be)		20	0x0014 (20)
8	Int8 (be)		124	0x7c (124)
9	Int16 (be)		1	0x0001 (1)
11	Int16 (be) | 0x8000	Remaining length (should be 294)		0x0126 (294)
13	Int16 (be)		8	0x0008 (8)
15	Int16 (be)		15	0x0010 (16)
17	Int8 (be)		0	0x00 (0)
18	Int16 (be)		49153	0x01c0 (448)
20	Int8 (be)		0	0x00 (0)
21	Int32 (le)		0x61637544 ”Duca”	0x61637544 (1633908036)
25	Int16 (be) | 0x8000	Remaining length (should be 280)		0x0118 (280)
27	MCS user-data/clientinfo			See **C2-0-0-0-3-0**
48	Tagged data			See **C2-0-0-0-3-1**

Table A.9: MCS Connect Initial, User data part

There are several values in the MCS Initial User data part (table A.9) that are unknown to us. They can be recognized by the lack of description. We simply use the same values as the native clients transfer over the network and it seems to work very well.

C2-0-0-0-3-0		MCS userdata/clientinfo		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int16 (le)	Client info tag	49153	0xc001 (49153)
2	Int16 (le)	Client info length	136 in rdesktop, 276 bytes remaining	0x00d4 (212)
4	Int16 (le)	RDP version	0x0001 for RDP4, 0x0004 for RDP5	0x0004 (4)
6	Int16 (le)		8	0x0008 (8)
8	Int16 (le)	Width		0x0320 (800)
10	Int16 (le)	Height		0x0258 (600)
12	Int16 (le)		51713	0xca01 (51713)
14	Int16 (le)		43523	0xaa03 (43523)
16	Int32 (le)	Keylayout		0x0000041d (1053)
20	Int32 (le)	Client build		0x00000a28 (2600)
24	Unicode string(16)	Hostname		SAURONS-_TOILET..
40	Int32 (le)		4	0x00000004 (4)
44	Int32 (le)		0	0x00000000 (0)
48	Int32 (le)		12	0x0000000c (12)
52	[unknown type]	Reserved data		See **C2-0-0-0-3-0-0**
75	ColorDepth (Int16 (le))	(client)		0xca01 (51713)(8 bits depth)
77	Int16 (le)		0	0x0001 (1)
79	[unknown type]	Remaining client data		See **C2-0-0-0-3-0-1**

Table A.10: MCS Connect Initial, User data part, client info part

C2-0-0-0-3-0-0		[unknown type] Reserved data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	RAW			00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
16	RAW			00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
32	RAW			00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
48	RAW			00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....

Table A.11: MCS Connect Initial, User data part, client info, part 1

C2-0-0-0-3-0-1		[unknown type] Remaining client data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	RAW			00 00 00 00 10 00 07 00 01 00 00 00 00 00 00 00 ....
16	RAW			00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
32	RAW			00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
48	RAW			00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
64	RAW			00 00 00 00 00 00 00 00 00 00 00 00 ....

Table A.12: MCS Connect Initial, User data part, client info, part 2

C2-0-0-0-3-1		Tagged data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Tagged datapart	Unknown		See **C2-0-0-0-3-1-0**
23	Tagged datapart	TAG_CLI_CRYPT		See **C2-0-0-0-3-1-1**
46	Tagged datapart	TAG_CLI_CHANNELS		See **C2-0-0-0-3-1-2**

C2-0-0-0-3-1-0		Tagged datapart Unknown		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int16 (le)	Tag		0xc004 (49156)
2	Int16 (le)	Length		0x000c (12)
4	[unknown type]	Data		See **C2-0-0-0-3-1-0-0**

C2-0-0-0-3-1-0-0		[unknown type] Data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	RAW			09 00 00 00 00 00 00 00 ....

C2-0-0-0-3-1-1		Tagged datapart TAG_CLI_CRYPT		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int16 (le)	Tag		0xc002 (49154)
2	Int16 (le)	Length		0x000c (12)
4	[unknown type]	Data		See **C2-0-0-0-3-1-1-0**

C2-0-0-0-3-1-1-0		[unknown type] Data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	RAW			01 00 00 00 00 00 00 00 ....

Table A.13: MCS Connect Initial, User data part, client info

C2-0-0-0-3-1-2		Tagged datapart TAG_CLI_CHANNELS		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int16 (le)	Tag		0xc003 (49155)
2	Int16 (le)	Length		0x002c (44)
4	RDP Channel data	Data		See **C2-0-0-0-3-1-2-0**

Table A.14: MCS Connect Initial, User data part, client info, part 1

C2-0-0-0-3-1-2-0	RDP Channel data Data			
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int32 (le)	Number of channels		0x00000003 (3)
4	Latin1 String(7 + nullchar)	Channel #0 name		rdpdr
12	RDP Channel Flags	Channel #0 flags		0x80800000 (2155872256) OPTION-_INITIALIZED, OPTION-_COMPRESS-_RDP,
16	Latin1 String(7 + nullchar)	Channel #1 name		cliprdr
24	RDP Channel Flags	Channel #1 flags		0xc0a00000 (3231711232) OPTION-_INITIALIZED, OPTION-_ENCRYPT-_RDP, OPTION-_COMPRESS-_RDP, OPTION-_SHOW-_PROTOCOL,
28	Latin1 String(7 + nullchar)	Channel #2 name		rdpsnd
36	RDP Channel Flags	Channel #2 flags		0xc0000000 (3221225472) OPTION-_INITIALIZED, OPTION-_ENCRYPT-_RDP,

Table A.15: MCS Connect Initial, User data part, client info, part 2

A.2.3 Multipoint Communication Service Protocol Setup, Connect Response

As an answer to the MCS Connect Initial, the servers send a MCS Connect Response. This as well is a packet filled with a lot of data. We have had to divide it into tables A.16 to A.26.

S2		TPKT from Server		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x05fe (1534)
4	TPDU			See **S2-0**
S2-0		TPDU		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPDU hdr length		0x02 (2)
1	Int8 (be)	TPDU packet type		0xf0 (240) Data
2	Int8 (be)	TPDU eot	128	0x80 (128)
3	MCS packet			See **S2-0-0**
S2-0-0		MCS packet		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	MCS Connect Response			See **S2-0-0-0**

Table A.16: MCS Connect Initial, TPKT, ISP, and MCS headers

S2-0-0-0		MCS Connect Response		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	BER tag 32614	MCS Conn response	32614	32614
2	VariableInt(2)	BER length		0x05f2 (1522)
5	BER tag 10	TAG-_RESULT	10	10
6	VariableInt(1)	BER length		0x0001 (1)
7	KLIInt(8)	Result Value		0x00 (0)
8	BER tag 2	INTEGER	2	2
9	VariableInt(1)	BER length		0x0001 (1)
10	KLIInt(8)	Connection id Value		0x00 (0)
'11	Domain Parameters	Target		See **S2-0-0-0-0**
30	MCS response userdata	User data		See **S2-0-0-0-1**

Table A.17: MCS Connect Response

S2-0-0-0-0		Domain Parameters Target		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	BER tag 48	Target	48	48
1	VariableInt(1)	BER length		0x001a (26)
2	BER tag 2	INTEGER	2	2
3	VariableInt(1)	BER length		0x0001 (1)
4	KLIInt(8)	Channels Value		0x22 (34)
5	BER tag 2	INTEGER	2	2
6	VariableInt(1)	BER length		0x0001 (1)
7	KLIInt(8)	Users Value		0x03 (3)
8	BER tag 2	INTEGER	2	2
9	VariableInt(1)	BER length		0x0001 (1)
10	KLIInt(8)	Tokens Value		0x00 (0)
11	BER tag 2	INTEGER	2	2
12	VariableInt(1)	BER length		0x0001 (1)
13	KLIInt(8)	Priorities Value		0x01 (1)
14	BER tag 2	INTEGER	2	2
15	VariableInt(1)	BER length		0x0001 (1)
16	KLIInt(8)	Throughput Value		0x00 (0)
17	BER tag 2	INTEGER	2	2
18	VariableInt(1)	BER length		0x0001 (1)
19	KLIInt(8)	Height Value		0x01 (1)
20	BER tag 2	INTEGER	2	2
21	VariableInt(1)	BER length		0x0003 (3)
22	KLIInt(24)	PDUsize Value		0xfff8 (65528)
25	BER tag 2	INTEGER	2	2
26	VariableInt(1)	BER length		0x0001 (1)
27	KLIInt(8)	Protocol Value		0x02 (2)

Table A.18: MCS Connect Response, Domain parameters

S2-0-0-0-1		MCS response userdata User data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	BER tag 4	Userdata length	4	4
1	VariableInt(2)	BER length		0x05cc (1484)
4	[unknown type]	T.124 data		See **S2-0-0-0-1-0**
25	MSVariable-Int(2)	Remaining length (remaining bytes: 1463)		0x05b5 (1461)
28	Tagged data	Tagged data		See **S2-0-0-0-1-1**

S2-0-0-0-1-0		[unknown type] T.124 data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	RAW			00 05 00 14 7c 00 01 2a 14 76 0a 01 01 00 01 c0 ....|..*.v......
16	RAW			00 4d 63 44 6e .McDn

Table A.19: MCS Connect Response, User data and T.124 data

S2-0-0-0-1-1 Tagged data Tagged data

Offset	Datatype	Description	Exp. val.	Value
0	Tagged datapart	TAG_SRV_INFO		See **S2-0-0-0-1-1-0**
23	Tagged datapart	TAG_SRV_SRV3		See **S2-0-0-0-1-1-1**
46	Tagged datapart	TAG_SRV_CRYPT		See **S2-0-0-0-1-1-2**

S2-0-0-0-1-1-0 Tagged datapart TAG_SRV_INFO

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (le)	Tag		0x0c01 (3073)
2	Int16 (le)	Length		0x0008 (8)
4	TAG_SRV_INFO	Data		See **S2-0-0-0-1-1-0-0**

S2-0-0-0-1-1-0-0 TAG_SRV_INFO Data

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (le)	RDP version		0x0004 (4)
2	Int16 (le)	Unknown	8	0x0008 (8)

S2-0-0-0-1-1-1 Tagged datapart TAG_SRV_SRV3

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (le)	Tag		0x0c03 (3075)
2	Int16 (le)	Length		0x0010 (16)
4	[unknown type]	Data		See **S2-0-0-0-1-1-1-0**

Table A.20: MCS Connect Response, User data, tagged data

S2-0-0-0-1-1-1-0 [unknown type] Data

Offset	Datatype	Description	Exp. val.	Value
0	RAW			eb 03 03 00 ec 03 ed 03 ee 03 00 00 ....

S2-0-0-0-1-1-2 Tagged datapart TAG_SRV_CRYPT

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (le)	Tag		0x0c02 (3074)
2	Int16 (le)	Length		0x059d (1437)
4	MCS response userdata cryptinfo	Data		See **S2-0-0-0-1-1-2-0**

Table A.21: MCS Connect Response, User data, contents of TAG_SRV_SRV3 and header of TAG_SRV_CRYPT

The contents of TAG_SRV_SRV3 is a mapping from RDP5 channels to MCS channels, although we do not parse or use that at the time of writing.

S2-0-0-0-1-1-2-0		MCS response userdata cryptinfo Data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int32 (le)	RC4 key size	1/40 bit, 2/128 bit	0x00000001 (1)
4	Int32 (le)	Encryption level	0/None, 1/Low, 2/Med, 3/High	0x00000002 (2)
8	Int32 (le)	Random salt len	32	0x00000020 (32)
12	Int32 (le)	RSA info len	1385	0x00000569 (1385)
16	[unknown type]	Server salt		See **S2-0-0-0-1-1-2-0-0**
43	[unknown type]	Cert header		See **S2-0-0-0-1-1-2-0-1**
70	Int32 (le)	CA Certificate length		0x000001c2 (450)
74	Certificate	(CA)		See **S2-0-0-0-1-1-2-0-2**
101	Int32 (le)	Certificate length		0x00000387 (903)
105	Certificate			See **S2-0-0-0-1-1-2-0-3**
132	[unknown type]	Remaining info		See **S2-0-0-0-1-1-2-0-4**

Table A.22: MCS Connect Response, User data, cryptographic information

S2-0-0-0-1-1-2-0-0 [unknown type] Server salt

Offset	Datatype	Description	Exp. val.	Value
0	RAW			51 f7 52 56 f6 1a ca d4 cd b0 84 e8 fc ad 41 62 Q.RV..........Ab
16	RAW			97 ad 82 86 8f 40 14 3e 72 c7 e4 de f7 1e cc b1 .....@.>r.......

S2-0-0-0-1-1-2-0-1 [unknown type] Cert header

Offset	Datatype	Description	Exp. val.	Value
0	RAW			02 00 00 00 02 00 00 00 ....

Table A.23: MCS Connect Response, User data, server salt, and certificate header

```
      Certificate (CA)
 Certificate:
Data:
    Version: 3 (0x2)
    Serial Number:
        01:9e:3f:0b:13:9e:19:50
    Signature Algorithm: sha1WithRSA
    Issuer: L=TCDEMO, CN=WIN2KTERM
    Validity
        Not Before: Jun 29 15:41:56 1970 GMT
        Not After : Jun 29 15:41:56 2049 GMT
    Subject: L=TCDEMO, CN=WIN2KTERM
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
        RSA Public Key: (512 bit)
            Modulus (512 bit):
                00:a4:84:b4:ea:78:ca:df:4b:58:f7:1c:bd:71:68:
                28:ce:78:1a:91:25:d0:6f:80:67:b7:d8:21:e5:c3:
                a7:a1:a6:0b:0b:6c:90:05:80:a5:9d:c7:bd:d2:d8:
                8b:cc:cf:34:0e:58:aa:87:86:92:12:8f:d2:aa:51:
                98:e1:9e:4b:73
            Exponent: 65537 (0x10001)
    X509v3 extensions:
        X509v3 Basic Constraints:
            CA:TRUE, pathlen:0
        1.3.6.1.4.1.311.18.8:
            H.H.W.3.3.B.3.B.X.6.H.9.V.W.Y.9.G.R.2.T.6.X.W.4.
            7.Y.W.8.K.9.Y.V.W.4.T...
Signature Algorithm: sha1WithRSA
    45:fa:79:30:e9:49:b9:42:59:fe:3c:2b:cf:85:e4:ed:fd:a6:
    76:34:a2:46:35:ea:5d:41:2d:27:e9:af:78:ba:a3:04:21:c5:
    08:2b:be:9b:c0:b8:02:74:4a:c1:77:5a:b3:5c:d4:49:14:bf:
    51:92:cf:bd:56:69:8e:d5:52:a7
```

Table A.24: X.509 certificate (CA)

```
      Int32 (le) Certificate length 0x00000387 (903)
      Certificate
 Certificate:
Data:
    Version: 3 (0x2)
    Serial Number:
        01:00:00:00:18
    Signature Algorithm: sha1WithRSA
    Issuer: L=TCDEMO, CN=WIN2KTERM
    Validity
        Not Before: Dec 31 23:00:00 1979 GMT
        Not After : Jan 19 03:14:07 2038 GMT
    Subject: CN=ncalrpc:WIN2KTER, L=ncalrpc:WIN2KTERM
    Subject Public Key Info:
        Public Key Algorithm: md5WithRSAEncryption
        Unable to load Public Key
    X509v3 extensions:
        1.3.6.1.4.1.311.18.4: critical
            ....
        1.3.6.1.4.1.311.18.2: critical
            M.i.c.r.o.s.o.f.t. .C.o.r.p.o.r.a.t.i.o.n...
        1.3.6.1.4.1.311.18.5: critical
            .0..............J.f.J.....3.d.2.6.7.9.5.4.-.
            e.e.b.7.-.1.1.d.1.-.b.9.4.e.-.0.0.c.0.4.f.a.3.
            0.8.0.d...3.d.2.6.7.9.5.4.-.e.e.b.7.-.1.1.d.1.
            -.b.9.4.e.-.0.0.c.0.4.f.a.3.0.8.0.d..............
        1.3.6.1.4.1.311.18.6: critical
            .0......D.W.I.N.2.K.T.E.R.M...5.1.8.7.9.-.2.7
            .0.-.2.1.7.9.3.3.3.-.7.6.6.9.9...T.C.D.E.M.O.....
        X509v3 Authority Key Identifier: critical
            0.....W.I.N.2.K.T.E.R.M.........
Signature Algorithm: sha1WithRSA
    09:c4:77:a8:84:09:4d:ff:ba:a1:1f:7b:3a:64:22:27:e4:1d:
    96:2f:1c:2c:28:ef:ce:ca:5e:ac:62:b6:81:98:37:21:53:2a:
    87:94:18:70:c5:9e:29:f5:d7:be:32:0d:54:11:dd:cc:38:e5:
    b4:8e:87:a5:17:c7:db:64:b5:c4
```

Table A.25: X.509 certificate (server)

The server certificate is used in setting up the encryption (see section 3.3). Note that the public key algorithm is set to md5withRSAEncryption in this example packet. That is incorrect because the value of the public key is really RSAEncryption. This is the reason we get an error, "Unable to load Public key". In the rdesktop code, we set the type to RSAEncryption and then load the key using OpenSSL [tea03] routines. The error does not seem to exist with all versions of Windows, though.

S2-0-0-0-1-1-2-0-4		[unknown type] Remaining info		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	RAW			00 00 00 00
				00 00 00 00
				00 00 00 00
				00 00 00 00
				....

Table A.26: MCS Connect Response, User data, unknown (padding?)

A.2.4 Multipoint Communication Service Protocol Setup, Erect Domain

C3 TPKT from Client

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x000c (12)
4	TPDU			See **C3-0**

C3-0 TPDU

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	TPDU hdr length		0x02 (2)
1	Int8 (be)	TPDU packet type		0xf0 (240) Data
2	Int8 (be)	TPDU eot	128	0x80 (128)
3	MCS packet			See **C3-0-0**

C3-0-0 MCS packet

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	MCS type		0x01 (1) EDRQ
1	EDRQ data			See **C3-0-0-0**

C3-0-0-0 EDRQ data

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (le)	SubHeight		0x0001 (1)
2	Int16 (le)	SubInterval		0x0001 (1)

Table A.27: MCS Erect domain

After the MCS Setup, the MCS Erect Domain request (EDrq) packet is transferred from the client to the server. The EDrq is defined in section 10.6 of [T.193b]. Its use in RDP is questionable–RDP does not use any of the domain management in T.125 so there is basically no need for telling the server what the height of the domain is since it is always going to be one. The EDrq packet is visualized in table A.27.

A.2.5 Multipoint Communication Service Protocol Setup, Attach User and MCS AUcf

C4 TPKT from Client

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x0008 (8)
4	TPDU			See **C4-0**

C4-0 TPDU

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPDU hdr length		0x02 (2)
1	Int8 (be)	TPDU packet type		0xf0 (240) Data
2	Int8 (be)	TPDU eot	128	0x80 (128)
3	MCS packet			See **C4-0-0**

C4-0-0 MCS packet

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	MCS type		0x0a (10) AURQ
1	[unknown type]			See **C4-0-0-0**

C4-0-0-0 [unknown type]

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**

Table A.28: MCS Attach User

S3		TPKT from Server		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x000b (11)
4	TPDU			See **S3-0**

S3-0		TPDU		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	TPDU hdr length		0x02 (2)
1	Int8 (be)	TPDU packet type		0xf0 (240) Data
2	Int8 (be)	TPDU eot	128	0x80 (128)
3	MCS packet			See **S3-0-0**

S3-0-0		MCS packet		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	MCS type		0x0b (11) AUCF
1	AUCF data			See **S3-0-0-0**

S3-0-0-0		AUCF data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	RT-SUCCESSFUL			See **S3-0-0-0-0**
19	Int16 (be)	User id		0x0006 (6)

S3-0-0-0-0		RT-SUCCESSFUL		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
off	RT-SUCCESSFUL			0

Table A.29: MCS Attach User Confirm

The MCS Attach User request, visualized in table A.28 from the client follows the EDrq and tells the server we want to add a user to the domain. The answer from the server comes in form of the MCS AUcf (“cf” probably as in “confirm”) packet, as visualized in table A.29. This packet contains the user id used in a lot of other requests.

A.2.6 Multipoint Communication Service Protocol Setup, Channel Configuration

Now follows one pair of CJRQ (Channel Join ReQuest) and CJCF (Channel Join ConFirm) for each channel in the connection. There are at least two channels–one user specific channel and one global graphics data channel. Also, there is one channel for each virtual channel in RDP5 for applications such as sound transfer, clipboard transfer, serial port redirection, and similar.

The CJRQ/CJCF pairs conclude the setup phase of the RDP.

An example CJRQ is in table A.30 and a CJCF is in table A.31.

A.3 The Session Phase

The session phase consists of two types of packets–either TPKT with MCS SDRQ or SDIN packets inside, or RDP5 “fastpath” data where the TPKT, ISO and MCS structures are stripped away in favor of a very simple header. The latter is a protocol defined by Microsoft. Documentation is not available to the public.

The contents of the packets from the server are mostly so called orders. Orders are instructions from the server to the client to draw some kind of graphics. The orders are defined in [MPP97]. We will not cover them here since they are out of the scope of this report. The orders are processed in `orders.c` in rdesktop.

The contents of the packets from the client are mostly keyboard and mouse events.

It is interesting to note that the native client sends TPKT packets only; no RDP5 packets are sent. The server however sends a large amount of RDP5 packets. We suspect that this has to do with a security problem in

C5 TPKT from Client

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x000c (12)
4	TPDU			See **C5-0**

C5-0 TPDU

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	TPDU hdr length		0x02 (2)
1	Int8 (be)	TPDU packet type		0xf0 (240) Data
2	Int8 (be)	TPDU eot	128	0x80 (128)
3	MCS packet			See **C5-0-0**

C5-0-0 MCS packet

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	MCS type		0x0e (14) CJRQ
1	CJRQ data			See **C5-0-0-0**

C5-0-0-0 CJRQ data

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (be)	User id		0x0006 (6)
2	Int16 (be)	Channel id		0x03ef (1007)

Table A.30: MCS Channel Join Request

S4 TPKT from Server

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	TPKT version	3	0x03 (3)
1	Int8 (be)	TPKT reserved	0	0x00 (0)
2	Int16 (be)	TPKT length		0x000f (15)
4	TPDU			See **S4-0**

S4-0 TPDU

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	TPDU hdr length		0x02 (2)
1	Int8 (be)	TPDU packet type		0xf0 (240) Data
2	Int8 (be)	TPDU eot	128	0x80 (128)
3	MCS packet			See **S4-0-0**

S4-0-0 MCS packet

Offset	Datatype	Description	Exp. val.	Value
0	Int8 (be)	MCS type		0x0f (15) CJCF
1	CJCF data			See **S4-0-0-0**

S4-0-0-0 CJCF data

Offset	Datatype	Description	Exp. val.	Value
0	RT-SUCCESSFUL			See **S4-0-0-0-0**
19	Int16 (be)	Initiator (user id)		0x0006 (6)
21	Int16 (be)	Requested		0x03ef (1007)
23	Int16 (be)	Channel id		0x03ef (1007)

S4-0-0-0-0 RT-SUCCESSFUL

Offset	Datatype	Description	Exp. val.	Value
off	RT-SUCCESSFUL			0

Table A.31: MCS Channel Join Confirm

the protocol where the checksum on the RDP5 packets became the same if the contents were the same (i.e., if you pressed the character 'f' twice, the packet would look exactly the same up to a certain point, rendering the encryption useless). The fix might have been to send TPKTs instead.

Data transferred using TPKTs have a channel defined that gives the destination on the receiver. This is used to differentiate for example graphics data and clipboard data–they use different channels.

A.3.1 RDP5 packets

When transferring graphics data in RDP5 mode, the server uses the RDP5 packet format that is different from the TPKT-TPDU-MCS structure. The new format has less overhead.

Basically, the new packet format begins with an 8 bit integer that has the value 0x80 (for an encrypted packet) or 0x00 (for an unencrypted packet) instead of the 0x03 used in TPKTs. This can be used to differentiate the two formats when processing them. After this header byte there is a length field of variable length, the crypto signature (if it is an encrypted packet) and then the encrypted data.

The encrypted data consists of subpackets where each subpacket has an 8 bit integer telling what type of subpacket it is, a 16 bit integer telling the length of the subpacket and then the data itself. The types of subpackets we know of are listed in table A.32.

A randomly chosen RDP5 packet can be seen in table A.33.

A.3.2 Clipboard packets

The clipboard transfer is done using a series of packets transferred on its own channel. The channel is allocated by the client both in the MCS Connect Initial packet (see table A.14) and using CJRQ/CJCF pairs in the channel setup (tables A.30 and A.31). In our case, the clipboard channel is channel number 1005.

The flags value in the clipboard packets are documented in [Cor03]. Basically, the least significant byte is 3 for a standalone operation, 1 for the first of several packets, 0 for a packet in the middle of several packets and 2 for the last packet of several. The next byte is set to 1 if the packet

Tag	Type	Comment
0x00	Orders	
0x01	Bitmap	
0x02	Palette	
0x05	NullSystemPointer	
0x06	DefaultSystemPointer	
0x07	MonoPointer	
0x08	Mouse Position	
0x09	ColorPointer	
0x0a	CachedPointer	
0x0b	Pointer	

Table A.32: RDP5 subpacket types

originates from the client, and to 0 if it originates from the server. The rest of the 32 bit integer is set to 0.

The Ptype0 and Ptype1 flags exist only in standalone packets or in the first of several packets. In table A.34 we have listed some possible values.

Clipboard Setup and Format Announce

In the beginning of the session, the server and client execute a clipboard handshake. This is a three step process, where the server first tells the client it wants to handshake (table A.35). The client responds with a list of formats available on the client side (if any) (table A.36 and A.37), and the server confirms this list with the packet in table A.38. The two last packets are also transferred each time either the server or the client have new clipboard data available. The part having new data sends the announce packet, and the other part sends an acknowledge packet.

For space-saving and clarity reasons, We have removed the TPKT, TPDU and MCS headers in the following packets.

The number of the format descriptions following the Remaining Length field is calculated by dividing the value of Remaining Length with 36, the size of a format description.

S20		RDP5 packet from Server		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int8 (be)	RDP5 start byte		0x80 (128), encrypted, 0 inputs
1	MSVariable-Int(2)	Packet length	1109	0x0455 (1109)
4	[unknown type]	Cryptsig		See **S20-0**
18	Palette			See **S20-1**
32	Int16 (le)	Partlen(?)		0x0308 (776)
34	[unknown type]	Part data		See **S20-2**
48	Orders			See **S20-3**
62	Int16 (le)	Partlen(?)		0x000c (12)
64	Orders	Part data		See **S20-4**
78	Bitmap update			See **S20-5**
92	Int16 (le)	Partlen(?)		0x0019 (25)
94	Bitmap update	Part data		See **S20-6**
108	Mouse pointer (b/w)			See **S20-7**
122	Int16 (le)	Partlen(?)		0x0111 (273)
124	[unknown type]	Part data		See **S20-8**
138	[unknown type]	Remaining data		See **S20-9**

Table A.33: A RDP5 packet

Ptype0	Ptype1	Comment
1	0	First handshake from server.
3	1	Acknowledge on our format announce.
3	2	Failed format announce. We resend the format announce here a few times.
2	0	Format announce from server.
5	1	Clipboard data.
4	1	Server request for our clipboard data

Table A.34: Possible values for Ptype0 and Ptype1

Data Transfer

For small clipboard data transfers, everything is transferred in one packet with the least significant byte in the flag set to 3. For larger transfers, the data is split into packets of at most 1600 bytes in size where the first packet has the flag byte set to 1, the middle packet has the flag bytes set to 0 and the last packet has the flag byte set to 2. The Remaining length field is set to the total length of the packets in all packets. The choice of 1600 bytes size for the packets is a bit odd given the fact that the Maximum Transfer Unit (MTU) of the most common network type (Ethernet) is 1500 bytes. Fragmentation will occur.

The data transfer begins with a request packet from the part that wants the data. Table A.39 lists such a packet, originating from the client.

The server responds with the clipboard data. Table A.40 lists such a packet.

S76-0-0-0 SDIN data

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (be)	Initiator (user id)		0x0001 (1)
2	Int16 (be)	Channel id		0x03ed (1005)
4	Int8 (be)	Flags		0xf0 (240)
5	MSVariable-Int(1)	Data length		0x0020 (32)
7	Int32 (le)	Flags(?)		0x00260008 (2490376)
11	RAW	Crypto signature		8 bytes of data
19	Clipboard data			See **S76-0-0-0-0**

S76-0-0-0-0 Clipboard data

Offset	Datatype	Description	Exp. val.	Value
0	Int32 (le)	Clpbrd data length		0x0000000c (12)
4	Channel data flags	Flags		0x00000003 (3) FLAG_FIRST, FLAG_LAST,
8	Int16 (le)	Ptype0		0x0001 (1)
10	Int16 (le)	Ptype1		0x0000 (0)
12	Int32 (le)	Remaining length		0x00000000 (0)
16	Int32 (le)	Unknown (Pad?)		0x0006ff24 (458532)

Table A.35: Clipboard data, Handshake, packet 1/3

C23-0-0-0		SDRQ data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int16 (be)	Initiator (user id)		0x0006 (6)
2	Int16 (be)	Channel id		0x03ed (1005)
4	Int8 (be)	Flags		0x70 (112)
5	MSVariable-Int(2)	Data length		0x00b0 (176)
8	Int32 (le)	Flags(?)		0x773f0008 (2000617480)
12	RAW	Crypto signature		8 bytes of data
20	Clipboard data			See **C23-0-0-0-0**

C23-0-0-0-0		Clipboard data		
Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int32 (le)	Clpbrd data length		0x0000009c (156)
4	Channel data flags	Flags		0x00000013 (19) FLAG_FIRST, FLAG_LAST,
8	Int16 (le)	Ptype0		0x0002 (2)
10	Int16 (le)	Ptype1		0x0000 (0)
12	Int32 (le)	Remaining length		0x00000090 (144)
16	Clipboard format description	#0		See **C23-0-0-0-0-0**
38	Clipboard format description	#1		See **C23-0-0-0-0-1**
60	Clipboard format description	#2		See **C23-0-0-0-0-2**
82	Clipboard format description	#3		See **C23-0-0-0-0-3**
104	Int32 (le)	Unknown (Pad?)		0x00000000 (0)

Table A.36: Clipboard data, Handshake, packet 2/3

C23-0-0-0-0-0 Clipboard format description #0

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int32 (le)	Numeric code		0x0000000d (13)
4	Unicode string(16)	Text representation		....

C23-0-0-0-0-1 Clipboard format description #1

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int32 (le)	Numeric code		0x00000010 (16)
4	Unicode string(16)	Text representation		....

C23-0-0-0-0-2 Clipboard format description #2

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int32 (le)	Numeric code		0x00000001 (1)
4	Unicode string(16)	Text representation		....

C23-0-0-0-0-3 Clipboard format description #3

Offset	**Datatype**	**Description**	**Exp. val.**	**Value**
0	Int32 (le)	Numeric code		0x00000007 (7)
4	Unicode string(16)	Text representation		....

Table A.37: Clipboard data, Handshake, packet 2/3

S77-0-0-0 SDIN data

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (be)	Initiator (user id)		0x0001 (1)
2	Int16 (be)	Channel id		0x03ed (1005)
4	Int8 (be)	Flags		0xf0 (240)
5	MSVariable-Int(1)	Data length		0x0020 (32)
7	Int32 (le)	Flags(?)		0x00260008 (2490376)
11	RAW	Crypto signature		8 bytes of data
19	Clipboard data			See **S77-0-0-0-0**

S77-0-0-0-0 Clipboard data

Offset	Datatype	Description	Exp. val.	Value
0	Int32 (le)	Clpbrd data length		0x0000000c (12)
4	Channel data flags	Flags		0x00000003 (3) FLAG_FIRST, FLAG_LAST,
8	Int16 (le)	Ptype0		0x0003 (3)
10	Int16 (le)	Ptype1		0x0001 (1)
12	Int32 (le)	Remaining length		0x00000000 (0)
16	Int32 (le)	Unknown (Pad?)		0x00010001 (65537)

Table A.38: Clipboard data, Handshake, packet 3/3

C107-0-0-0 SDRQ data

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (be)	Initiator (user id)		0x0006 (6)
2	Int16 (be)	Channel id		0x03ed (1005)
4	Int8 (be)	Flags		0x70 (112)
5	MSVariable-Int(1)	Data length		0x0024 (36)
7	Int32 (le)	Flags(?)		0x3a700008 (980418568)
11	RAW	Crypto signature		8 bytes of data
19	Clipboard data			See **C107-0-0-0-0**

C107-0-0-0-0 Clipboard data

Offset	Datatype	Description	Exp. val.	Value
0	Int32 (le)	Clpbrd data length		0x00000010 (16)
4	Channel data flags	Flags		0x00000013 (19) FLAG_FIRST, FLAG_LAST,
8	Int16 (le)	Ptype0		0x0004 (4)
10	Int16 (le)	Ptype1		0x0000 (0)
12	Int32 (le)	Remaining length		0x00000004 (4)
16	Int32 (le)	Requested format code		0x0000000d (13)
20	Int32 (le)	Unknown (Pad?)		0x00000000 (0)

Table A.39: Clipboard data request

S182-0-0-0 SDIN data

Offset	Datatype	Description	Exp. val.	Value
0	Int16 (be)	Initiator (user id)		0x0001 (1)
2	Int16 (be)	Channel id		0x03ed (1005)
4	Int8 (be)	Flags		0xf0 (240)
5	MSVariable-Int(1)	Data length		0x002e (46)
7	Int32 (le)	Flags(?)		0x00260008 (2490376)
11	RAW	Crypto signature		8 bytes of data
19	Clipboard data			See **S182-0-0-0-0**

S182-0-0-0-0 Clipboard data

Offset	Datatype	Description	Exp. val.	Value
0	Int32 (le)	Clpbrd data length		0x0000001a (26)
4	Channel data flags	Flags		0x00000003 (3) FLAG_FIRST, FLAG_LAST,
8	Int16 (le)	Ptype0		0x0005 (5)
10	Int16 (le)	Ptype1		0x0001 (1)
12	Int32 (le)	Remaining length		0x0000000e (14)
16	[unknown type]	Clipboard data		See **S182-0-0-0-0-0**
39	Int32 (le)	Unknown (Pad?)		0x00000100 (256)

S182-0-0-0-0-0 [unknown type] Clipboard data

Offset	Datatype	Description	Exp. val.	Value
0	RAW			66 00 6f 00 6f 00 62 00 61 00 72 00 00 00 f.o.o.b.a.r...

Table A.40: Clipboard data sent from server

Appendix B

Security Problems With Microsoft's RDP Client Implementations

During our investigations of RDP we found that none of the available RDP clients from Microsoft verified the public key of the server. This is a rather serious security flaw as it opens up for Man In The Middle (MITM) attacks such as the one we used to find information about the protocol sent over the network.

It is worth to note that even if Microsoft would rewrite their client and verify host keys, reverse-engineering would still be possible using other techniques such as disassemblers and debuggers. Depending on the implementation of host key verification, MITM attacks might still be possible as well.

This appendix will discuss the security problems in Microsoft's RDP implementation, found during the thesis work. We will also briefly mention the licensing modes available in RDP, since the licensing mode in use has implications for the encryption model used.

B.1 The Problem

While setting up the session key as described in section 3.3.2 it is crucial that we use the correct public key. If we use the wrong key, the server cannot decrypt what we encrypted. However, if there is a party in the middle doing a MITM attack as described in 2.3.1 that wants to decrypt our conversation, it can replace the server's public key with its own, sending the new key down the line to the client. The client will then encrypt its session key with this key, and send it to the MITM host which decrypts it and then sends it further to the server, using the correct key of the server.

The only way you can know you are talking to the real server and not to a malicious third party in the middle is by verifying that the server's public key is the correct one, belonging to the server. There are several different methods for doing so.

B.1.1 Verifying the Host key - At the First Connection

One way of adding security in this context is to store the server's public key (the so-called host key) the first time a connection is made. This way, subsequent connections will be protected from MITM attacks. By ensuring that the first connection is made on a secure network (if they exist, see section 3.4.4), a full protection from MITM attacks can be ensured.

The clients from Microsoft do not use this technique.

B.1.2 Verifying the Host Key - Manually

Another way of adding security is to verify the host key by creating a digital fingerprint. This is an algorithm that creates a short binary number that can be compared on the server and client side using some other secure communcations channel.

This method is often used in combination with storage of the host key after the first connect. That is, at the first connection you are given the opportunity to verify the host key. At subsequent connects, the host key is already verified and stored client-side.

The clients from Microsoft do not provide any way to find out what fingerprint the server you are connecting to has. There is also no convenient nor documented way to find out what fingerprint a server has from the server's side.

B.1.3 Verifying the Host Key - Using a CA

Another way of adding security is to have a chain of certificates. This way you can verify **one** key from a so called Certificate Authority and then trust all keys that are signed with this key. A Certificate Authority (CA) is a party that the client trusts that issues certificates for servers. This is the technique used for many Internet sites using the Secure Sockets Layer (SSL), commonly known as HTTPS. Another name for this concept is PKI for "Public Key Infrastructure" [Tan02].

The X.509-based structure sent from the server in RDP5 has two keys, where one (the actual server key) is signed by the other (the CA key). However, there is no way to verify the CA key.

To summarize, none of the Microsoft clients we tried verified the host key of the server, allowing for undetected MITM attacks at any time.

B.2 Reporting a Security Flaw In a Responsible Way

Since we believe it is good Internet manners to report security problems in order to make the Internet a safer place, we decided to report this problem. There are different ways of reporting vulnerabilities of this type.

One way is to notify only the vendor and trust that they fix the problem and notify their customers in a fast way. On the other end of the scale is full disclosure with publication of tools that make it easy to exploit the vulnerability.

We chose a middle way where we notified Microsoft of the problem and then gave them some time to react. Since they did not react in a satisfying way (actually, they did not seem to understand the problem at all) we went public with the information by announcing the problem on BUGTRAQ [Sec03a], a well known mailing list for this type of information. We did not provide the tools we had (rdpproxy and pparser.py) since we did not want less serious individuals on the Internet to get their hands on the tools needed too easily.

For a list of frequently asked questions about vulnerability reports, see [Sec03b].

B.3 Licensing Modes

Since the licensing mode of the server affects the encryption setup, we briefly discuss it here.

A Terminal Server can be in one of two different modes. Either it is in Application Server Mode, which requires licensing per connecting machine or per user (the latter only in Windows 2003), or in Remote Administration Mode, a feature that allows remote administration of Windows servers available in some Windows versions. The Remote Administration mode has a limit of two concurrent connections per machine.

We discovered that the encryption setup is different in the two. In the former, a X.509-based public key structure is used, in the latter a raw RSA structure.

B.4 License Negotiation

During RDP setup, license negotiation takes place. This directly affected our work with implementing RDP5 support for rdesktop.

The license negotiation used in RDP is quite complex. Basically, each device that connects to a Terminal Server is granted a license. That is, the licenses are granted per machine connected, not per user. Note that in Windows 2003 Server, a “per user” licensing mode has appeared, but we have not been able to investigate this.

Now, in earlier versions of Terminal Services, the license was granted before the user logged in. This caused problems since you could easily run out of licenses. In later versions, temporary licenses are granted at first connect. If a valid user logs in, a permanent license is granted the next time a connection is made.

Regardless of this, the license negotiation at the protocol level is also very complex. To start with, there is extra encryption added to the license tokens. In RDP4, this meant the license tokens where encrypted once, but in later versions they are encrypted with the transport encryption as well, meaning two different encryptions on top of each other. This was a problem when listening to the traffic between client and server using rdpproxy, since the RC4 state got invalid when trying to decrypt and then encrypt the license data. Therefore, we had to keep track of the RC4 state to be able to correctly pass through the license negotiation between the client and the server.

Index

Avdelning, Institution Division, Department ADIT, Dept. of Computer and Information Science 581 83 Linköping LINKÖPINGS UNIVERSITET	**Datum** Date 2004-03-05

Språk Language ☐ Svenska/Swedish ☒ Engelska/English ☐ ______	**Rapporttyp** Report category ☐ Licentiatavhandling ☒ Examensarbete ☐ C-uppsats ☐ D-uppsats ☐ Övrig rapport ☐ ______	**ISBN** — **ISRN** LITH-IDA-EX–04/082--SE **Serietitel och serienummer** Title of series, numbering **ISSN** —
URL för elektronisk version		

Titel Reverse-Engineering och Implementation av RDP 5

Title Reverse-Engineering and Implementation of the RDP 5 Protocol

Författare Author Erik Forsberg

Sammanfattning
Abstract

The Remote Desktop Protocol (RDP) is a protocol for accessing Microsoft Windows Terminal Services. The protocol provides remote desktop services, meaning a graphical desktop is sent to the client, and user input (keyboard and mouse events) are sent to the server, all over a bandwidth-narrow channel. The protocol is used by thin clients, i.e. clients with small resources, to reach servers in a server-based computing environment.

There is an RDP-client called Rdesktop, written for Unix-like operating systems. It has an X Window System graphical user interface and provides access to Terminal Servers from the Unix environment. Rdesktop, however, only supports version 4 of the RDP. The current version of RDP (August 2003) is 5.

Documentation of RDP can be acquired from Microsoft, but not without signing a non-disclosure agreement (most often referred to as "NDA"). This means it is not possible to create a program with the source code available without breaking the agreement. Therefore, implementation of open source RDP clients must be preceded by reverse engineering activities.

In this report we describe how we reverse engineered version 5 of RDP and how we implemented support for it in rdesktop. We have implemented support for basic RDP5 as well as support for clipboard operations between the X Window System and Microsoft Windows.

Among the future work on rdesktop that will be possible to investigate as a result of this thesis work are support for sound redirection, disk drive redirection as well as support for more clipboard formats.

Nyckelord
Keywords
Reverse Engineering, Network Analysis, Software Engineering, Network Security, Remote Desktop Protocol.