Studienarbeit

Implementing the NTLM Secure Service Provider for Wine

Kai Blin

Betreut von Prof. Dr-Ing. Wilhelm G. Spruth und Dipl-Inform. David Gümbel

University of Tübingen
Wilhelm Schickard Institute for Computer Science
Department of Computer Engineering

Hiermit erkläre ich, daßich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

. . . .

Ort, Datum, Unterschrift

Abstract

The Windows API allows applications to quickly retrieve user credentials using a special API called Secure Service Provider Interface (SSPI). This paper presents how the NTLM Secure Service Provider was implemented in Wine.

Specifically, it explains the use of `ntlm_auth` to parse NTLMSSP packets, how `ntlm_auth` is called from within Wine and what additional code was written to handle inter-process communication with `ntlm_auth`.

It also presents on how the NTLM Secure Service Provider integrates with the Secure Service Provider Interface and how that interface integrates with Wine.

I would like to thank Andrew Bartlett of the Samba team for his patience on showing me around `ntlm_auth`.

Special thanks go to Wine's Juan Lang for teaching me how to write code the Wine way and for keeping me motivated even when my patches were not accepted on the first few tries.

Contents

1 Introduction

This section will provide a short description of the topics covered.

1.1 Authentication

The most important question authentication has to answer is ”Who are you?”. This is not only focused on authenticating users who want to log on to a computer, but it is also brought up when authenticating against a remote server, where you want to make sure that you are not sending confidential information to an untrusted remote site.

Most of these authentication models are based on trust. Somewhere in the model, there has to be a single instance that is trusted by the user. In many environments, this instance is called the *login* or *password server*.

In the area of computing, several methods to answer the ”Who are you?” question have evolved. Some are more secure than others, but almost always additional security comes at the cost of usability.

1.1.1 Shared secret

The most commonly used method for authentication is the *shared secret* method. It is based on you knowing something the other side knows, too. Commonly this is a password that is stored on the computer and entered by the user. By comparing the stored password with the one on file the login server can decide if the user knows the shared secret. If the passwords match, the user is authenticated.

This method, if used standalone, has the problem of not offering a simple way for the user to make sure he really is authenticating against the correct server. As the user is sending his shared secret to the server, a malicious server posing as the user’s login server could capture the shared secret and subsequently use it to impersonate the user on the real login server.

As the confidentiality of the shared key is vital for the security of this authentication method, it is important to not send the shared key over the wire for remote authentication purposes. Otherwise it might be stolen and used to impersonate the user on the system he wanted to log in to.

The bonus of this method is that it is relatively common, so users are familiar with entering passwords to authenticate themselves. If client/server authentication is handled by a different method, it is relatively safe to send a hashed version of the secret to the login server to authenticate the user.

1.1.2 One time pads

As shared secret authentication is prone to *replay attacks*, where attackers use a captured shared secret to impersonate a user, one time pads are taking the shared secret to the next level. The basic idea is the same but for one time pads, a shared secret is used only once.

This of course comes with the cost of more difficult logistics. The user needs a list of one time passwords which needs to be replaced whenever he is out of passwords. The login server can easily store multiple passwords and keep track of the passwords already used up, but needs a secure method for getting newly generated password lists to the user.

A common application for this are TAN numbers for online banking, where the possible damages avoided make up for the inconvenience of this method.

1.1.3 Biometric identification

Shared secrets are often lost, forgotten or can be stolen. Biometric characteristics are part of every person. They also cannot be easily stolen. Thus biometric characteristics are ideal to be used in place of a shared secret to authenticate users. The server has the characteristics on file and compares them to the characteristics measured using sensors.

There are a couple of disadvantages. Some biometric characteristics change from time to time, like voices. The system has to allow for some changes in the voice or it will stop somebody with a sore throat from using the system. Being less specific of course increases the risk of incorrectly identifying and authorizing an unauthorized person.

Some biometric characteristics can be used to obtain more information about the user authenticating than needed. Fingerprints and retina patterns could be used to find out about a possible criminal history or medical problems. Thus they pose an intrusion into the user's privacy that cannot be easily dismissed.

As the sensory equipment and computing power needed to take and compare those characteristics, in addition to the problems mentioned above, biometric identification is only used for high-security areas.

1.1.4 Asymmetric keys

Using an asymmetric key authentication scheme is a pretty secure way of authenticating.

Unfortunately, this requires that every public key of an entity that has to be authenticated needs to be stored for every entity that wants to authenticate. On the server side, this is not too much of a problem, but users do not want to remember dozens of different, complicated public keys, as well as their private key.

Keys can be stored on a token card for users, but every time a server is added, a private key is changed or revoked, the token cards need to be updated, too. Also, all systems interacting with the users need readers for the token cards. Additionally, token cards can be lost or stolen, so at least the user's private key needs to be secured by using an additional authentication method. Thus it is not really convenient to use for user authentication.

It is convenient to use for establishing the client/server trust relationship that makes shared secret user authentication more secure.

1.2 Single sign-on

On a modern computer system there are many different applications that may want to authenticate the user. The easiest way to do this is by having one set of user name and password for every program. From the usability point of view, this is rather bad, and usually leads to passwords written on post-it notes sticking to the screen.

The solution to this is single sign-on, where the user authenticates only once and other applications use the cached credentials to handle their authentication needs. Different programs and protocols have evolved for this. On of the most popular of those is Kerberos (see [NT94]), which was developed at the MIT.

1.3 NTLMSSP

NTLMSSP, the NT/LanManager security service provider, was developed by Microsoft to provide network authentication services for Windows NT 4.

The details of the protocol were never released by Microsoft, but the Samba team reverse engineered and documented the protocol (see [Gla03]). Recently, the web page about NTLM was updated by the Samba team, so the version used for this project is given in the appendix.

For Windows 2000, a new version of NTLM was introduced, called NTLMv2, and the old version was renamed to NTLMv1. NTLMv2 differs to NTLMv1 only in the cryptographic algorithms used. The basic protocol handshake is the same and up until Windows 2003 Server, all newer implementations would fall back to NTLMv1 if the other side of the client/server communication does not support NTLMv2.

The NTLM protocol makes use of a variety of hash functions and stream ciphers to ensure message integrity and handle encryption. NTLMv1 uses MD4 (see [Riv92a]) and ARCFOUR (see [KT99]). NTLMv2 uses MD4, MD5 (see [Riv92b]), ARCFOUR and HMAC-MD5 (see [KBC97]).

1.4 Wine

Wine is an intermediate layer mapping Windows API functions to POSIX API functions.

Figure 1 shows a Linux program and a Win32 program running on a Linux system. The Linux program directly calls POSIX API functions to access functionality offered by the operating system. A Win32 program running in Windows would make similar calls to the Win32 API. Because the Win32 API and the POSIX API are incompatible and Win32 uses a different binary format, Win32 programs cannot run on Linux systems directly.

This is where Wine comes in. It offers the Win32 API functions Win32 applications expect to be there and either calls the appropriate POSIX API functions or handles the calls itself. For all the Win32 program cares, it is running on a Win32 system.

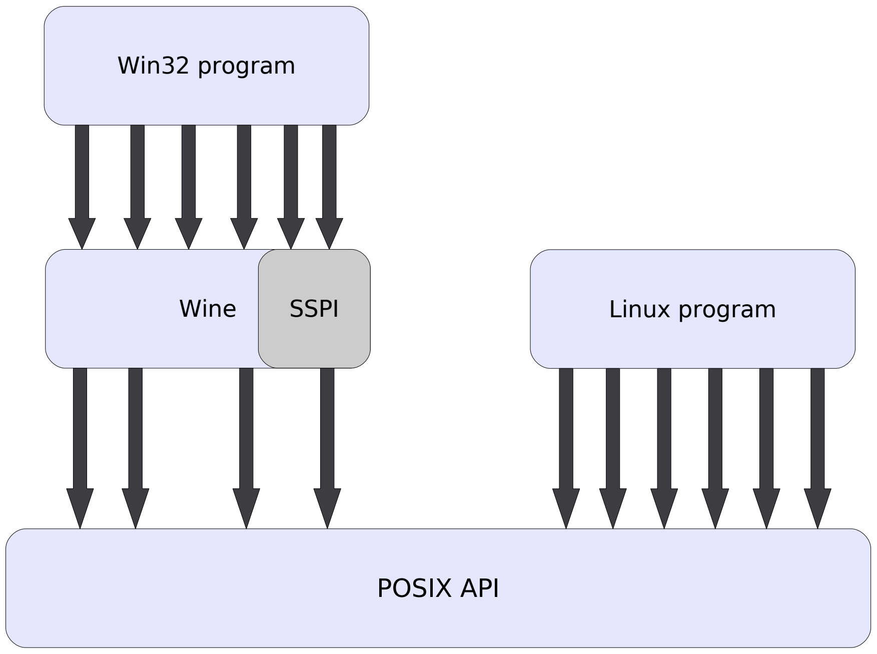

Figure 1: Win32 programs on Linux using Wine.

2 Materials and methods

There are two major libraries that bundle the task of negotiating and selecting authentication protocols and abstracting the protocol used. One of these libraries, the Secure Service Provider Interface, is Windows based and the other, GENSEC works on POSIX systems.

2.1 SSPI

SSPI was designed by Microsoft to have a unified programming interface for application protocols that offers authentication, message integrity and message encryption, without any changes to the original application protocol (see [Cor99]).

2.1.1 Overview

SSPI uses a plug-in based approach with individual providers being loadable as separate DLLs. An exception to this are the builtin providers, on Windows 2000 these are Negotiate, NTLM, MSN and Kerberos. A plug-in that also is installed with Windows installations since Windows 95 is the SCHANNEL provider that offers SSL encryption. For an overview, see *Figure 2*.

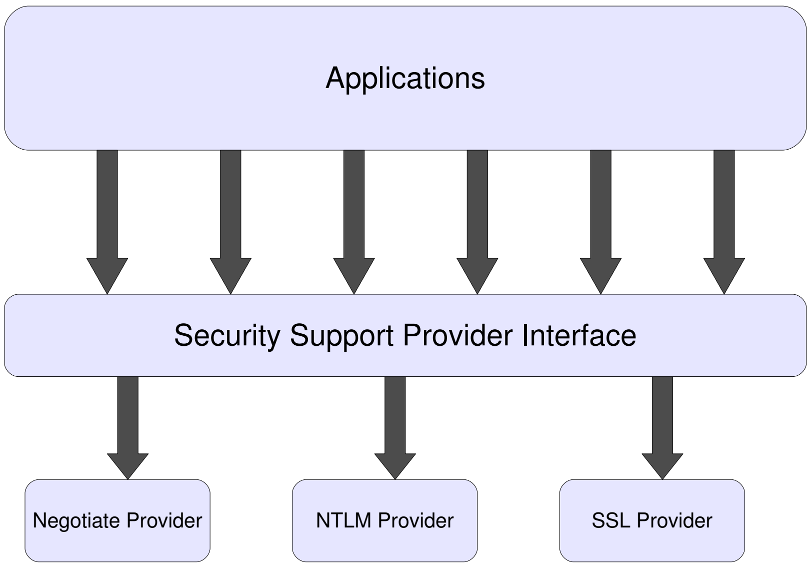

Figure 2: Secure Service Provider Interface overview

To allow applications to stay independent of the actual plug-ins installed on the system, SSPI offers a couple of capabilities that different modules, called Secure Service Provider (SSP) have. Applications can request cryptographic capabilities from SSPI and let SSPI handle the plug-in to choose. Usually this will be the Negotiate provider that implements SPNEGO (see [BP98]).

SPNEGO just negotiates which actual security module to use, it usually picks between Kerberos and NTLM. From the programmer's point of view, the security protocol chosen is not important.

2.1.2 The NTLM handshake

NTLM uses a three way handshake to set up the trust relationship. First the client notifies the server of it's wish to authenticate. The server sends an authentication challenge to the client that the client needs to encrypt with a shared secret, usually the user's password. The client performs the operation and replies with the authentication response. If the response is correct, the client is authenticated. Optionally, client and server also agree on a session key that can be used for message encryption and message integrity checks later.

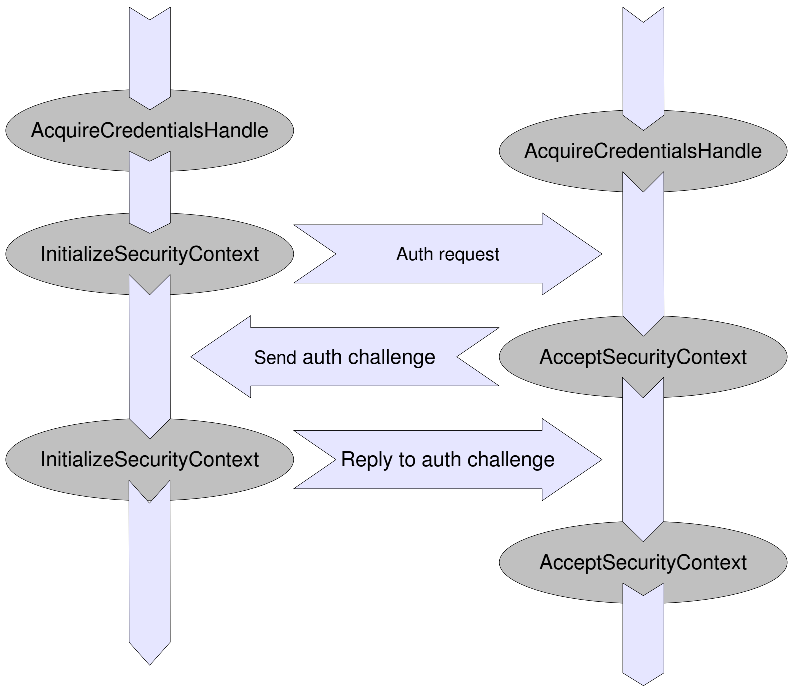

Figure 3: NTLM handshake

2.1.3 Important API functions

Figure 3 shows that the NTLM handshake uses three function calls in the SSPI API. Those are AcquireCredentialsHandle, InitializeSecurityContext and AcceptSecurityContext. More functions are involved in the complete process, those are listed in Appendix A.

AcquireCredentialsHandle Used for both the client and server side, this function creates a credential handle, an opaque data structure dependent on the user's credentials. Credential handles can be reused for multiple connections.

InitializeSecurityContext On the client side, this function creates and establishes a security handle. Security handles are opaque data structures bound to a specific connection and cannot be reused.

AcceptSecurityContext This function is used on the server side to create a security handle for the connection with the client. It cannot be reused.

2.2 GENSEC

A good implementation of the NTLM protocol is found in the GENSEC library (see [Bar05]) that is part of the Samba software package. It provides a clean API and can handle all of the functionality SSPI offers. However, a couple of issues prevent it's use for a Wine

DLL. GENSEC is coupled to other Samba code in a rather complex manner, using e.g. debugging or memory allocation functions from other parts of the Samba code. This makes including the library into Wine code unwieldy. An equally big problem is the licensing situation. GENSEC is licensed under the GNU General Public license (GPL), as are most of the dependencies. Wine is licensed under the GNU Lesser General Public license (LGPL) to allow linking the winelib functions to proprietary programs. Due to the character of the GNU GPL, it is not possible to include GNU GPL code into GNU LGPL code.

While the Samba team offered to release the GENSEC library under the GNU LGPL, the effort involved in resolving all the dependencies to other Samba code and reworking the memory management was judged as too high to be worthwhile.

2.3 ntlm_auth

`ntlm_auth` is a program from the Samba software package that was originally designed as a plug-in to the Squid proxy server to offer NTLM authentication to Internet Explorer clients. Squid offers a defined protocol (see [CC01]) to communicate with plug-ins on an inter-process communication basis. By using this form of communication, no code is mixed and the software licenses used are irrelevant. The protocol also makes it easy to add plug-ins to Squid.

To allow for easy network communication with the plug-ins, binary data is encoded using the Base64 encoding (see [Jos03] and [Jos06]). `ntlm_auth` uses the Base64 codec included with Samba. For the Wine side, a Base64 codec was written in the process of this work.

Currently, there are two versions of Samba that provide a version of `ntlm_auth` that can be used with this implementation. At the time of writing, this is Samba 3.0.25 and Samba 4.0tp3. Depending on the version used, different steps must be taken to create the user database for SSPI in server mode.

3 Results

My work on implementing the NTLM Secure Service Provider in Wine consisted in creating three source files, `ntlm.c`, textttdispatcher.c and `base64_codec.c`, about 2000 lines of code. *Figure 4* shows how the Wine NTLM SSP works in detail.

3.1 ntlm.c

The main logic for the NTLM SSP is contained in this file. `ntlm.c` registers the provider with the SSP interface and sets up the function lookup tables SSPI uses. As the Win32 API supplies a lot of the functions in an ANSI and an unicode variant, all the ANSI functions convert the strings passed to unicode and call the appropriate unicode function to avoid code duplication.

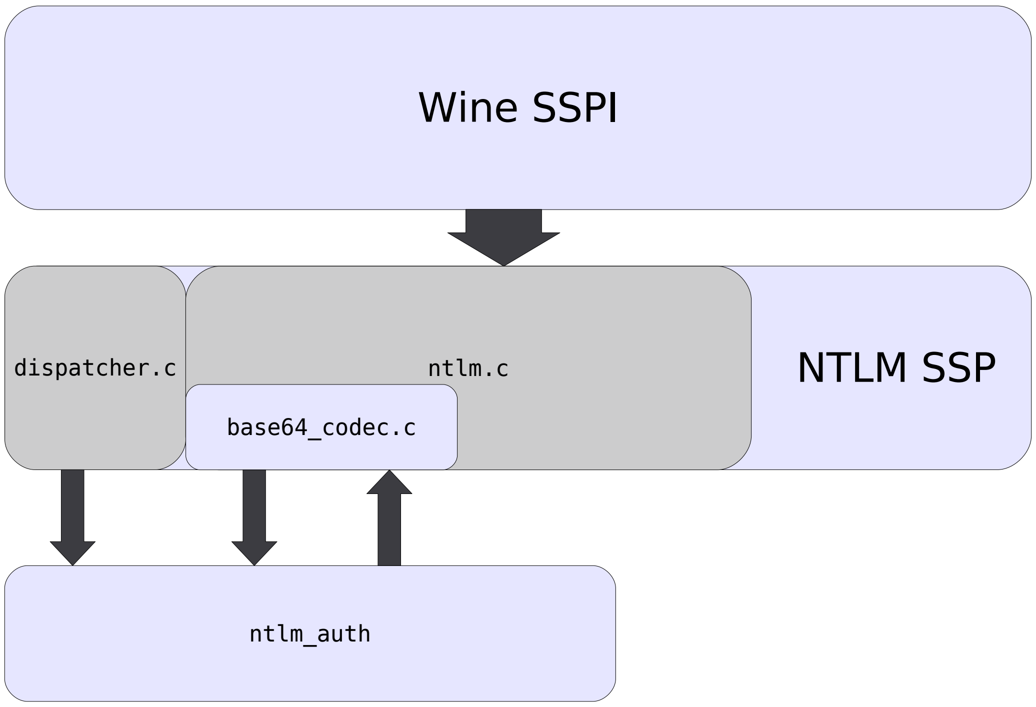

Figure 4: Wine's NTLM Secure Service Provider in detail

`ntlm.c` sets up the credential and session handles, as well as giving out information about the properties of the NTLM SSP. For the actual NTLM handshake, it will make use of functionality provided by `ntlm_auth` via functions in `dispatcher.c`, converting to and from Base64 using functions in `base64_codec.c` if needed.

Notable functions in `ntlm.c` are *AcquireCredentialsHandle*, *InitializeSecurityContext* and *AcceptSecurityContext*.

Later, `ntlm.c` was extended to also support signing and encrypting of the packets passed to it, but this is beyond the scope of this text.

3.1.1 AcquireCredentialsHandle

As this function is used for both client and server credentials, both cases need to be handled here. Windows supports a third option, where the credential can be used on another server. As the user handling in Wine is still very basic, it was decided not to implement this functionality just yet.

Depending on the credential requested, `ntlm_auth` is started in either server or client mode. To authenticate users in server mode, the user needs to exist in the Samba user database. If Samba 3 is used, `ntlm_auth` needs access to the privileged named pipe to talk to `winbindd`, which in turn accesses the password database. This rather complex setup is not needed for Samba 4, where `ntlm_auth` can access the user database directly.

The likely use case of Wine's SSPI is client side NTLM, with the server running either Samba or Windows. More complex features of server side NTLM are not possible on Wine yet, due to the lacking user management.

3.1.2 InitializeSecurityContext

This function is called repeatedly by SSPI to create client packets to be sent to the server until the authentication handshake is complete. In the case of NTLM, two packets are sent. *InitializeSecurityContext* does not keep track of the state it is in but rather decides on what the input parameters are.

Depending on the state, different commands are sent to `ntlm_auth`. In the initial state, the command sent is **YR**, which resets `ntlm_auth` to it's initial state and acquires a new NTLM authentication request packet. To avoid being asked for a password after that, a Base64 encoded password is sent before **YR** is sent. The reply is **YR** with a Base64 encoded NTLMSSP auth request. The reply is decoded and passed back to the calling application, which then has to send the packet to a NTLM server. The server replies with an authentication challenge packet. This is Base64 encoded and passed to `ntlm_auth` again, using the **TT** command. `ntlm_auth` replies with, depending on the version, **KK** or **AF** and the Base64 encoded authentication reply. This again is decoded and passed to the caller. If the server accepts the authentication reply, the authentication was successful.

It is important to note that even if the server does not accept the authentication reply, e.g. if the password was wrong or the user does not exist, *InitializeSecurityContext* will still return `SEC_E_OK`. The only way to find out if the authentication actually worked is to attempt to access the resource one authenticated for.

3.1.3 AcceptSecurityContext

Called for the server side of SSPI, this function responds to authentication requests by generating an authentication challenge. To do so, the input buffer is Base64 encoded and sent to `ntlm_auth` using the **YR** command. The **TT** reply is then decoded and passed back to the calling application, which then has to send it back to the client. On the second call, the authentication response from the client is Base64 encoded and passed to `ntlm_auth` using the **KK** command. `ntlm_auth` evaluates the authentication response by checking against the Samba user database and either succeeds, returning **AF** or fails, returning **NA**. If the authentication succeeded, the user is now authenticated.

3.2 dispatcher.c

`ntlm_auth` needs to be run in a different process and its input and output streams need to be redirected to file streams. Functions called from `ntlm.c` write to and read from these file streams.

fork_helper forks a new process, diverts the input and output descriptors and executes `ntlm_auth`. In the old process, it sets the input and output descriptors accordingly.

run_helper sends a text string to `ntlm_auth` and reads back the reply, dynamically growing the memory for the character buffer used. It also performs some basic error handling for `ntlm_auth` errors.

cleanup_helper frees the communication buffer and closes the stdin and stdout descriptors for `ntlm_auth`, thus closing it and reaping the child process.

check_version checks the version of `ntlm_auth`. This is used by `ntlm.c` to make sure to only register the NTLM SSP when the version of `ntlm_auth` is recent enough to support the used functionality.

3.3 base64_codec.c

This source file implements a simple but fast Base64 encoder/decoder. The two main functions are *encodeBase64* and *decodeBase64*. Both do what their name implies.

4 Discussion

While there are still a lot of tasks left to be able to use single sign-on in Wine, first steps in this direction have been taken. Recent releases of `ntlm_auth` support credential caching, so it is no longer necessary to repeatedly query users for passwords. Eventually, Wine will implement a Local Security Authority (LSA), making both Wine-based credentials caching and server side possible.

At the time of writing, the Wine SSPI is used to authenticate Remote Procedure Calls (RPC) using the NTLM SSP for Microsoft Outlook 2003 against a Microsoft Exchange 2003 server. Other software using the same RPC calls will work as well, but I am not aware of any use cases.

The implementation of client-side NTLM authentication using a wrapper for `ntlm_auth` is likely to be included in Heimdal Kerberos GSSAPI implementation to add NTLM support. There are plans to use it to add NTLM support to the Cyrus implementation of SASL (see [Mye97] and [MZ06]), too.

List of Figures

Acronyms

GPL	General Public license
LGPL	Lesser General Public license
LSA	Local Security Authority
RPC	Remote Procedure Calls
SSP	Secure Service Provider
SSPI	Secure Service Provider Interface

References

[Bar05] A. Bartlett. GENSEC - designing a security subsystem. Technical report, Samba Team, April 2005. http://us1.samba.org/samba/news/articles/gensec-white-paper.pdf, accessed 2005-09-05.

[BP98] E. Baize and D. Pinkas. The Simple and Protected GSS-API Negotiation Mechanism. RFC 2478 (Proposed Standard), December 1998. Obsoleted by RFC 4178.

[CC01] F. Chemolli and R. Collins. The Squid-NTLM helper protocol, 2001. http://devel.squid-cache.org/ntlm/squid_helper_protocol.html, accessed 2005-09-07.

[Cor99] Microsoft Corporation. The Security Support Provider Interface. Technical report, Microsoft Corporation, 1999. http://www.microsoft.com/windows2000/docs/sspi2000.doc, accessed 2005-09-12.

[Gla03] E. Glass. The NTLM Authentication Protocol and Security Support Provider. Technical report, jCIFS, 2003. http://davenport.sourceforge.net/ntlm.html, accessed 2005-09-05.

[Jos03] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 3548 (Informational), July 2003. Obsoleted by RFC 4648.

[Jos06] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[KT99] K. Kaukonen and R. Thayer. A Stream Cipher Encryption Algorithm ”Arcfour”, December 1999.

[Mye97] J. Myers. Simple Authentication and Security Layer (SASL). RFC 2222 (Proposed Standard), October 1997. Obsoleted by RFC 4422, updated by RFC 2444.

[MZ06] A. Melnikov and K. Zeilenga. Simple Authentication and Security Layer (SASL). RFC 4422 (Proposed Standard), June 2006.

[NT94] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, 1994.

[Riv92a] R. Rivest. The MD4 Message-Digest Algorithm. RFC 1320 (Informational), April 1992.

[Riv92b] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

A Documentation of the Functions

AcceptSecurityContext See section 2.1.3.

AcquireCredentialsHandle See section 2.1.3.

CompleteAuthToken According to [Cor99], this needs to be called if InitializeSecurity-Context returns SEC_I_COMPLETE_AND_CONTINUE or SEC_I_COMPLETE_NEEDED. This never seems to happen for NTLM.

DecryptMessage Decrypts messages encrypted by the EncryptMessage function.

DeleteSecurityContext Once a session is finished, this function clears the old session keys and deletes the session context handle.

EncryptMessage Encrypts data passed to the function using the ”arcfour” cipher.

FreeCredentialsHandle This function deletes the credential handle which contains information about the user. In Wine, this calls on `dispatcher.c` to clean up and terminate the `ntlm_auth` helper process.

InitializeSecurityContext See section 2.1.3.

MakeSignature Sign a message using a combination of HMAC(MD5) or CRC32 and ”arcfour”.

QueryContextAttributes Get information about different properties of the current session context.

VerifySignature Verify that a packet signature is valid.