VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

UNIFIED NETWORK AUTHENTICATION FOR LINUX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL ZŮNA

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

JEDNOTNÁ SÍŤOVÁ AUTENTIZACE PRO LINUX

UNIFIED NETWORK AUTHENTICATION FOR LINUX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — PAVEL ZŮNA
AUTHOR

VEDOUCÍ PRÁCE — Ing. JOZEF MLÍCH
SUPERVISOR

BRNO 2010

Abstrakt

Tato práce se zabývá návrhem a implementací řešení pro jednotnou síťovou autentizaci pro operační systém Linux založeného na integraci systémových démonů WinBind a SSSD. Cílem je navrhnout takové řešení, které umožní autentizaci Linuxových klientů do domén spravovaných adresářovými službami Windows Active Directory a domén spravovaných adresářovými službami dostupnými na Linuxu současně. První dvě kapitoly seznámí čtenáře s autentizačními mechanizmy a technologiemi, které se pro tyto účely používají na operačních systémech Windows a Linux. Třetí kapitola se zabývá jádrem práce a vysvětluje rozhodnutí učiněná při návrhu implementovaného řešení. Samotná implementace ja pak popsána v kapitole čtyři. Poslední kapitoly popisují experimenty a testování pro vybrané případy užití s návrhy a popisem možných rozšíření do budoucna.

Abstract

This thesis discusses the design and implementation of an unified network authentication solution for the Linux operating system based on the integration of WinBind and SSSD system daemons. The goal is to be able to authenticate Linux clients against multiple domains based on different platforms. In the first two chapters, readers are introduced to authentication mechanisms and related technologies used in Windows and Linux based computer network infrastructures. The third chapter is focused on the core of this work and discusses decisions made during the design phase. Implementation details are described in chapter four. The last part of the thesis describes experiments and tests for selected use cases along with ideas for future improvements.

Klíčová slova

bezpečnost, autentizace, adresářové služby, LDAP, Active Directory, Linux, WinBind, SSSD, NSS, PAM

Keywords

security, authentication, directory services, LDAP, Active Directory, Linux, WinBind, SSSD, NSS, PAM

Citace

Pavel Zůna: Unified Network Authentication for Linux, diplomová práce, Brno, FIT VUT v Brně, 2010

Unified Network Authentication for Linux

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jozefa Mlícha

....
Pavel Zůna
May 25, 2011

Poděkování

Na tomto místě bych chtěl poděkovat všem, kteří mi s touto prací pomohli ať už přímo nebo nepřímo. Zejména svému vedoucímu Jozefu Mlíchovi za věcné připomínky k textu, Simovi Sorceovi a Stephenovi Gallagherovi za exkurzi kódem SSSD, Dmitri Palovi za možnost na tomto projektu pracovat, Víťovi, Kačence a dalším kamarádům za podporu.

Contents

Chapter 1

Introduction

Computer networks are a key component in the information infrastructure of any modern day organization. Unconnected computers are becoming an ever shrinking minority used only for special applications. Virtually the only place where we can find computers not connected to any network today are embedded systems and even there, completely standalone machines are becoming rare.

Over the years as computers were becoming cheaper and more common as everyday tools, the size of computers networks grew accordingly. As they were becoming larger and more data and resources were made available through them, it was necessary to take security into account. Access control was nothing new and standalone multi-user systems implemented them, but it had to be extended over the whole network and adequately represent the organizational structure of organizations, which led to the creation of software solutions for centralised management of networks and sub-networks.

Such centralised solutions have existed on both Windows and Linux platforms (the platforms this thesis is concerned about) for a long time and are constantly being improved as requirements on network management grow. Unfortunately, the existing solutions targeted at one of the two platforms are mostly uncompatible with solutions targeted at the other one. Selected solutions for both platforms with emphasis on user account management and their potential interoperability are discussed in chapter 2.

There are ways for Windows clients to enroll in a Linux domain and vise-versa, but there is no standard solution for unified network authentication of Linux clients in the sense of authenticating against a Windows based and Linux based domain simultaneously. The goal of this thesis is to stand up to the challenge of creating such a solution using existing facilities. The existing facilities in question are the NSS and PAM services available on most if not all Linux distributions and the WinBind and SSSD system daemons implemented on top of them. All of the mentioned technologies are described in details in chapter 3.

The core of this thesis is dedicated to the design and implementation of an integration scheme for WinBind and SSSD, but it wouldn't be complete without proper testing of the resulting implementation for target use cases. Chapter 4 and chapter 5 are focused on designing the integration and on implementation details respectively.

It is a personal goal of me, the author of this work, to get the results „out there“ and eventually make the life of network administrators of mixed platform domains easier. Chapter 6 discusses the results of this work, what goals have been met and possible enhancements and follow up work in the future.

Chapter 2

Managing Network Accounts

The core of this thesis is focused on the problem of client network authentication. Nevertheless, we're going to start by looking at the server side as it is crutial to understand the protocols and technologies against which we're going to authenticate.

Before we get to more sophisticated solutions used in practice. Let's outline the simplest possible scheme for authenticating users used on the historically first computer networks. Each computer had the information about accounts stored in files (e.g. `/etc/passwd`) locally. In the case of an update to this information, a new version of the file would be distributed to all computers.

The scheme described in the previous paragraph might be acceptable for small networks consisting of a few nodes. However, as the network grows and the number of nodes increases, distribution of user account information becomes very expensive and unmanageable. It also creates an environment were synchronization errors can occur and lead to potentially exploitable security threats. This and other difficulties like the unability to create fine grained security policies have led to the developement of new solutions with centrally managed user accounts.

Let this serve as a brief introduction to the topic of managing user accounts along with other shared information on computer networks. We're now going to explore the most commonly used technologies created for this purpose, although the scope will be limited to Linux and Windows platforms only. Other platforms are left out as they are of no interest for the goals of this thesis. Some of the solutions described in the following sections are becoming outdated, but knowledge of them might be helpful to readers in a better understanding of current designs and practices.

2.1 Network Information Service

Network Information Service (NIS) is one of the first protocols that emerged for the purpose of sharing network wide information. It follows the client-server model where clients retrieve shared information from a directory service.

In this context, a directory service is a software system that stores, organizes and provides access to information stored in directory, which is a database optimized for reading entries and name value pairs associated with them.

Apart from user accounts, NIS can be used to maintain information about hostnames/ machine addresses, security information, mail information, Ethernet interfaces and network services [3]. Note that NIS predates DNS.

Being one of the directory service pioneers, NIS has s few drawbacks especially when it comes to security. There is no support for encryption and some of the mechanisms it uses are unsecure by design. For example, clients had to do a broadcast to find a running NIS server on the network; a weak spot for man-in-the-middle attacks as anyone could answer the broadcasted queries and impersonate the real server.

An improved version of the protocol named NIS+ later developed. Although similar in name and purpose, the implementations are completely different. Unlike NIS, NIS+ uses a hierarchical directory, that can/should be serviced by multiple servers. The primary server is known as master and other (backup) servers are known as replicas or slaves [3]. Both types hold copies of the directory data. All updates have to be committed to the master server and is then propagated to replicas in increments. NIS+ is also based around Secure RPC where servers must authenticate clients and vice-versa [10].

The protocol was developed by Sun Microsystems for their own Solaris operating system. It was very successful and got licenced to virtually all other Unix vendors. Altough NIS and it's successor are still in use today they aren't developed or supported by any major vendors anymore in favor of new more modern and secure directory services such as LDAP.

Prior to the release of Solaris 9, Sun Microsystems has announced it intends to drop NIS+ in future releases and started shipping tools to migrate NIS+ data to LDAP [2].

2.2 LDAP Based Solutions

Similar to NIS in certain aspects, LDAP is another client-server directory service protocol. It is based on the X.500 specification for directory services [23].

LDAP can be used to store and retrieve arbitrary data (including binary). Its directory is organized as a tree of entries called directory information tree (DIT). An entry consists of a set of attributes that can have one or more values. Each entry in DIT has a unique identifier refered to as a distinguished name (DN). DNs are constructed from two parts. First part is taken from the most relevant attribute and is called the relative destinguished name (RDN). Second part is the parent entry DN.

Attributes in LDAP can also hold references to other LDAP server making it possible to have a directory spanning multiple servers.

The contents of entries in DIT are governed by the so called schema - a special entry stored outside of the directory tree. It defines attribute types and object classes. Object classes define what attributes must or may be stored in an entry of a certain type. All entries in LDAP must have an objectClass attribute. An example of an user account is shown on figure 2.1.

The protocol comes in two version currently being used. Version 2 (LDAPv2) was superseded by version 3 (LDAPv3) in 1997, which added support for extensibility, improved security and better alignment with the latest X.500 specification [20].

Thanks to its universal design, extensibility and platform independence, LDAP popularity grew rapidly since it being published and the protocol has become a de facto industry standard of directory service protocols.

LDAP supports SSL encryption, but it doesn't have a native authentication mechanism. It does, however, support the Simple Authentication and Security Layer (SASL) and is often coupled with the Kerberos protocol.

```
dn: cn=John Doe,ou=people,dc=example,dc=com
cn: John Doe
givenName: John
sn: Doe
telephoneNumber: +1 888 555 6789
telephoneNumber: +1 888 555 1232
mail: john@example.com
manager: cn=Barbara Doe,dc=example,dc=com
objectClass: inetOrgPerson
objectClass: organizationalPerson
objectClass: person
objectClass: top
```

Figure 2.1: Example of user entry in LDAP

2.3 Kerberos

While Kerberos isn't a network management system, but rather an authentication protocol, it still deserves its own section in this chapter, because of its popularity and pairing with many directory services.

It's behind the scope of this article to explain the Keberos protocol in details, but because of its immense popularity when it comes to securing LDAP, readers should be familiar with its fundamentals and most imporatant advantages.

Kerberos uses as its basis the symmetric Needham-Schroeder protocol. It makes use of a trusted third party named a key distribution center (KDC), which consists of two logically separate parts. First part is an authentication server (AS) and a ticket granting server. The KDC maintains a database of secret keys. Each entity on in the domain secured by Kerberos (called a realm in Kerberos terminology) shares a secret key known only to itself and the KDC. Knowledge of this key servers to prove identity. When two entities on the network want to communicate, they ask KDC to generate an encrypted session key they can use for secure interactions [14].

The main feature of Kerberos is that entities on the network have to authenticate only once with the authentication server and access to all other entities is negotiated transparently with the tiget grantig server. The principle of authenticating only once is called single-sign on. The main advantage of it are that users have to remember only one password and are entering it in only one place when logging in into the network.

One disadvantage of Kerberos is that it hasn't been standardized and various existing implementations use incompatible APIs. Fortunately, many (if not all) Kerberos flavors can be translated through the Generic Security Services Application Program Interface (GSS-API) to procedures that SASL can understand.

2.4 FreeIPA

FreeIPA is a fully integrated security information solution. It combines a LDAP directory service (389 Directory Server, formerly known as Fedora DS), MIT implementation of the Kerberos protocol, certificate server (Dogtag), DNS (Bind) and NTP for Kerberos ticket synchronization on networks spread over different time zones.

It offers a plugin-extensible management framework with tools providing a higher level of abstration over its directory schema. Client can manage information stored with FreeIPA through the framework a special XML-RPC API or by using direct LDAP calls. The schema included by default is a compatible extension to the schema used in 389 Directory Server.

As 389 Directory Server supports multi-master replication, network security architects and administrators deploying FreeIPA can make user of this feature as well.

Since version 2.0, a web based user interface with self-service capabilities for the end users (users of the network without administrative privileges), although it's still in BETA phase at the time writing this thesis.

The project is developed by Red Hat as an alternative to current LDAP and Kerberos based systems. It's goal is to become the standard solution when it comes to network management. Integration with Microsoft Active Directory Domain Services (discussed later in this chapter) is planned for version 3.0, which is currently in the design phase.

2.5 Microsoft NT Directory Services

NT Directory Services (NTDS) is the commercial name for a network directory service used in networks made up of Windows clients. A network of clients sharing the directory service provided by NTDS is called a Windows Server Domain. In such a domain, the directory database resides on server configured as domain controllers. All domain controllers respond to security authentication requests, but only one of them can be configured as the Primary Domain Controllers (PDC). All other domain controllers are setup as Backup Domain Controllers (BDC). BDC are read-only and all updates must be propagated through the PDC. In other words, NTDS doesn't support multi-master replication. An existing BDC can be readily promoted to PDC in case of the previous PDC being unavailable for some reason mitigating this disadvantage [17].

Clients can communicate with domain controllers only by using the proprietary Microsoft RPC, which is a modified version of The Open Group's DCE/RPC standard. Communication in an NT domain is secured using the NT LAN Manager (NTLM) suite of Microsoft's security protocols that provides authentication, integrity and confidentiality of the transmited information [4].

Since the release of Windows 2000 Server, NT Directory Services are now deprecated in favor of the more sophisticated Active Directory Domain Services, but backwards compatibility with the older system is still maintained.

2.6 Microsoft Active Directory Domain Services

With the introduction of Windows 2000 Server, the previously discussed NT Directory Services were replaced with Active Directory Domain Services (ADDS). It isn't just an incremental version update, but a completely new system.

The Primary/Backup Domain Controller model of NTDS wasn't good enough for new constantly growing networks and was superseded with a new one supporting multi-master replication. All domain controllers in an Active Directory domain can propagate updates over the network [17].

Communication between client and domain controllers has also been improved and uses Microsoft Message Passing Interface (MS MPI), which is a proprietary implementation of the MPI-2 standard designed for message passing between high performance computing nodes. Active Directory also supports both LDAPv2 and LDAPv3 protocols and allows direct access to the directory using LDAP commands [17].

Active Directory is secured with Microsofts own, again proprietary, implementation of Kerberos and therefore provides single sign-on for user access to managed network resources.

It replaces the old NTLM suite of protocols used in NT Directory Services, but is still used by Active Directory is some special cases of inbound authentication such as client authenticating using an IP address (instead of hostname) or if Kerberos ports are restriced by firewall rules [17].

Other features of ADDS include, but are not limited to: DNS services, security policies, access control, integration with other Microsoft products (such as Exchange) and graphical management tools such as the Microsoft Management Console.

2.7 Interoperability

After reading the previous section of this chapter, the reader should be familiar with the most common systems for managing network user accounts. While both Linux and Windows solution use similar technologies, it isn't easy to integrate them. The main causes are different implementations of these technologies and different representation of user account data.

Integration of Windows network services with Unix has been a long term goal of the Samba community project. Samba version 3 and higher can integrate with NT Directory Services and act as a Primary Domain Controller. It can also become part of an Active Directory domain, but only as a member. The ability of Samba acting as a Active Directory domain controller is planned for version 4 [12].

As mentioned before in this chapter, the FreeIPA project is aiming for Active Directory integration in its upcoming major release.

Microsoft has its own answer to Windows and Unix interoperability called Windows Services for UNIX (SFU). SFU's focus isn't the integration of Windows and UNIX directory services, but provides a Unix subsystem and other parts of a full Unix environment on Windows machines. It does however provide NIS server capabilities linked with Active Directory and tools for transparent handling of NFS mount points as Windows shared directories and vise-versa [6].

Chapter 3

Network Authentication on Linux

At this point, we've been throught the systems used for managing network user accounts on Windows and Linux networks and will now „switch to the client side“. As this chapter name suggests, you won't find any information here, that relates to the authentication of Windows clients, as it is outside of the scope of this thesis.

We're going to look at standard facilities developed over the years available to Linux clients for authenticating against the systems and technologies described in the previous chapter.

Our main area of interest is the authentication of Linux client in mixed network environments and the possiblities of authenticating against multiple domains, or against multiple directory services if you prefer.

3.1 Name Service Switch

Name Service Switch (NSS) is a facility in the Unix family of operating systems that enables these systems to define a variety of sources for common configuration databases and name resolution mechanisms [9].

Configuration of NSS usually takes place in the file `etc/nsswitch.conf`. This configuration file enables system administrators to set a list of modules of name resolution services for different types of objects such as users, groups, machines (hosts) and possible others [15].

Name resolution services that want to be configurable with NSS have to implement special modules, that NSS can load and execute when the service is requested. These modules are implemented as static libraries. NSS find the appropriate modules using an enforced naming conventions. Names of NSS modules are prefixed with the string „nss_“.

NSS is implemented as part of the standard C library, so that calls to functions such as getent calls the appropriate NSS module [5]. This assures that existing applications don't have to be changed and recompiled when NSS configuration changes or a new NSS module is loaded.

One major disadvantage of NSS is that its configuration allows only a list with static order of name resolutions services to be defined for the supported objects. When NSS handles a name resolution request, it calls the first module in this list and only after it fails, the next module in line is invoked. This can be a problem if a user needs to authenticate against more than one domain.

3.2 Pluggable Authentication Modules

Pluggable authentication modules (PAM) is a similar mechanism to NSS in concept. When a PAM aware application is started, it activates its attachement to the PAM API. Configuration files are read and the appropriate module routines are invoked.

PAM needs to be configured independently for every program or service that wants to take advantage of its features. Each consumer of the PAM framwork needs to have its configuration stored in a separate file under `/etc/pam.d`. For example, the SSH daemon settings can be found in `/etc/pam.d/sshd`.

Unfortunately, PAM isn't problem free. Configuration requires extensive knowledge of the handled security protocol and application it applies to. Another problem is that limitations of the PAM API don't allow a PAM module to request Kerberos service tickets from a Kerberos Key Distribution Center (KDC) and only allow it to get ticket granting tickets [8]. Therefore, users have to re-enter their password if they wish to authenticate with Kerberos through a PAM enabled applications, that isn't specifically coded for it, which hinders the single-sign on principle.

3.3 Authentication Against LDAP

Authentication of client machines against LDAP is widely supported by most if not all Unix distributions including Linux. Open source modules that communicate with LDAP are available for both NSS and LDAP.

Both modules read the same configuration file named `/etc/ldap.conf` on most systems [1]. The most important configurable values include: LDAP protocol version, the server hostname or IP address and port, the distinguished name of the directory root, a search filter to identify user entries and encryption settings for PAM.

As most solutions involing LDAP are secured using Keberos, because the default authentication mechanism in LDAP is consider weak and doesn't offer the advantages of single-sign on, pam_ldap is rarely used in favor of a specialized Keberos PAM modules. In the case of the MIT implementation Kerberos V prevalent on Linux, the module pam_krb5 is used. pam_krb5 uses its own configuration file named `/etc/krb5.conf` [8].

3.4 WinBind Daemon

WinBind is a daemon that allows its host system to become a full member of an NT or Active Directory domain. Once this is done, the host system will see users and groups in the Windows domain as if they were native to the host system. This is achieve throught WinBind provided NSS and PAM modules allowing name resolution and authentication with a Windows domain controller. The result is that whenever a program on the host system asks to look up a user or group name, the query will be redirected by NSS to WinBind and consequently to the Windows directory service. The whole process is completely transparent.

A subset of Microsoft RPC procedures is implemented by WinBind to communicate with NT domain controllers and Microsoft MPI to do the same with Active Directory domain controllers [13]. This means that WinBind uses the native communication protocols which results in the host system being undistinguishable from a Windows box in the target domain.

WinBind maintains a database called `winbind_idmap.tdb` in which it stores mappings between UNIX user and group IDs (UID/GID) and NT security IDs (SID; used both for users and groups). This mapping is used only for users and groups that do not have a local UID/GID. WinBind allocates a portion of the range of possible IDs and maps NT SID into it. Instead of the `winbind_idmap.tdb` file, an idmap backend can be specified and the mapping is retrieved using it instead. For this purpose an LDAP backend is available [13].

Both NT and Active Directory systems can generate a lot of user and group name lookups. To reduce the bandwidth cost of these lookups, WinBind uses a caching scheme based on the sequence number of requests generated by domain controllers [13]. This sequence number is incremented every time account information is modified in the directory and is called SAM. Lookup results returned by domain controllers are cached by WinBind along with SAMs. If the cache expires, the current SAM is requested from a primary domain controller. In case the requested SAM doesn't match the one stored locally, the cached information is discarded a new up to date information is requested.

The main use case of WinBind is for organizations that have an existing NT or Active Directory domain infrastructure into which they wish to put Linux workstations and servers.

WinBind is part of the Samba community project.

3.5 System Security Services Daemon

System Security Services Daemon (SSSD) is another daemon similar in function to WinBind. It also provides NSS and PAM modules and handles the authentication against directory services on the network.

SSSD implements a generic interface for authentication. Requests against different directory services are handled by service backends. Backends for communicating with LDAP and FreeIPA are available. The FreeIPA backend implements the IPA XML-RPC and enables native mode communication independent of the underlying LDAP schema. Using the LDAP backend, SSSD can communicate with Active Directory since it supports the LDAP protocol, but special configuration is needed because of the uncompatible schema. It also isn't the native communication protocol for Active Directory and is therefore susceptible to entry format changes in the ADs directory service.

Kerberos is well supported by SSSD. It can automatically renew tickets and also works around the inability of PAM to get service tickets from key distribution centers.

For the same reasons as WinBind (expensive directory lookups in terms of bandwidth), SSSD also implements a caching mechanism. Unlike WinBinds caching scheme, it doesn't rely on any specific features such as SAM sequence numbers and was designed to be more universal in order to support third party service backends.

Because of its importance for the capability to resolving authentication information of the host system, SSSD features an indepedently running daemon service called monitor. The role of this monitor service is to periodically check if SSSD is running and responding. If for some reason SSSD needs to be restarted, the monitor is responsible for handling this task.

SSSD is a young project developed since late 2008 by Red Hat. It was originally developed for the Fedora Linux operating system, but it is open source and has been designed with portability in mind from the very beginning. Other distributions such as Debian, Ubuntu or Gentoo have packages of SSSD available in their repositories. SSSD is the default authentication provider in Fedora 14 and upcoming releases.

Chapter 4

Integrating WinBind and SSSD

With introductory chapters over, we're ready to get to the core of the problem this thesis is trying to solve.

The goal is to have a simple solution for authenticating Linux clients against multiple domains in mixed environment networks. The solution should take advantage of facilities already in place needs to be transparent to the end user.

This thesis proposes the integration of the WinBind daemon and the SSSD daemon with the following traits:

- SSSD is the only authority for authenticating users
- WinBind acts as a gateway between SSSD and Windows directory services
- There is only one cache: SSSD and WinBind caches are unified
- The integration is transparent to the end user

Another solution would be to design and implement a completely new backend for SSSD, that would handle requests directed at NT and AD domain controllers using native RPC calls the same way WinBind does. However this would require a lot of new code with functionality already available to us in WinBind. It would take a considerably larger amount of time and would be very expensive. WinBind has been in developement for several years and has a time-tested mature code base. It's being actively developed by the Samba team, which is also an advantage, because they're working closely togheter with Red Hat on common AD integration solutions for Linux based operating systems. All of the mentioned reasons contributed to the decision of integrating WinBind with SSSD.

Some readers with more in depth knowledge of the subject this work is dealing with might object the NSS and PAM responders of WinBind and SSSD can coexist and both work on the same host system. The same would be true for currently existing solutions for LDAP identity retrieval and authentication solutions that connect to NSS and PAM. However, it is the whole purpose of SSSD to aleviate system administrators from having to manage several configuration files and setting up the corresponding daemons independently on each client of the network.

To meet the required traits listed at the start of this chapter, this work proposes the creation of a new SSSD backend with providers for identity, authentication, password changes and possibly security policy enforcement. NSS and PAM are going to send requests to the generic SSSD responders, which will automatically forward them to the newly implemented providers if they cannot be satisfied from SSSD internal cache. Each provider will handle its tasks by querying WinBind if necessary. The high level diagram in figure 4.1 depicts the proposed solution.

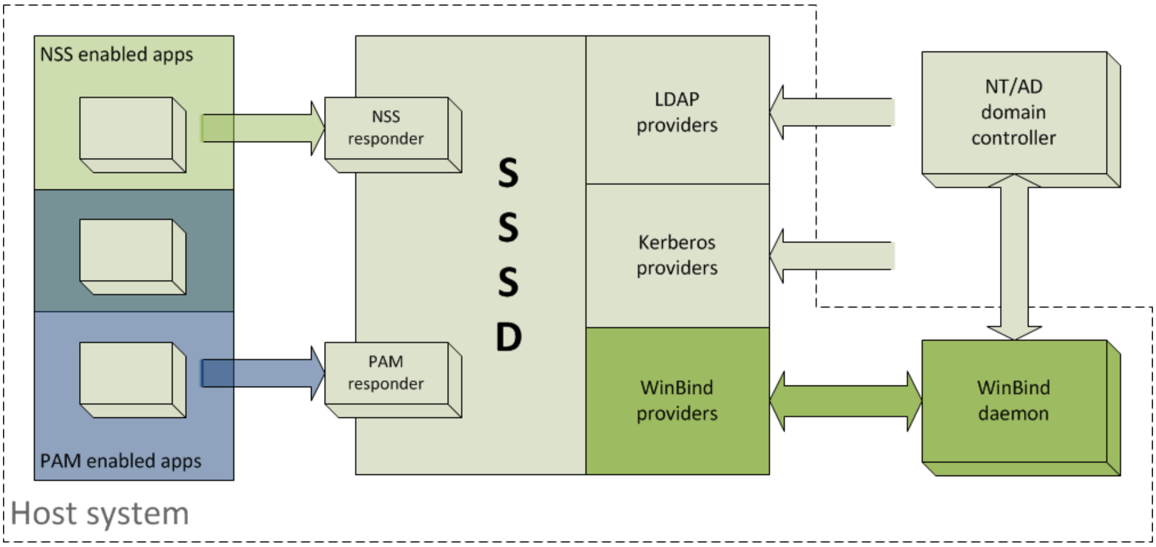

Figure 4.1: WinBind and SSSD integration: high level diagram

4.1 NSS and PAM responders

At this point, after reading section 3 of this paper, readers should be familiar with the purpose and basic functionality of both NSS and PAM. Now, we're going to look a bit deeper and discuss the sturcture of and how NSS and PAM responders are constructed. We already know that both are extended using static libraries called modules. These modules must implement a strictly defined interface described in the following paragraphs. Let's take a look at NSS first.

NSS responder

User and group identity information (NSS can also handle hostnames, netgroups, etc., but these are out of the scope of this work) are retrieved from the standard C library using the `getpwnam` and `getgrid` functions respectively. Both are then forwarded to NSS central routing mechanism handled by the `nsdispatch` function. All of these calls are defined in the header file `nsswitch.h`. The routing mechanism sequentially queries modules as they are written in the configuration file. If the module contains a handler for the requested operation, it is called and if it fails, the next module in row gets queried. The exception to this rule is enumeration where all modules are queried idependently if they support it [11].

PAM responder

The architectures of NSS and PAM are very similar. However, PAM is more complex as it provides more functionality. This funcitonality is split between four types of modules. The primary purpose of PAM is to do authentication, but it also provides for account and session management, and changing authentication tokens (such as passwords). If you're creating a new authentication mechanism, you don't have to define all of the supported module types, because some of them might be common to another mechanism already in place. Available module types are described shortly in table 4.1 [22].

Type	**Purpose**
Account	Account management Has user access at this time or on this console?
Auth	Authentication Is user who he claims to be?
Password	Changing authentication tokens Usually a password, finger print, ID card, ...
Session	Performing tasks when user starts/ends a session. Displaying last login, mail, mounting directories, ...

Table 4.1: PAM module types

This section was more of a detour for the purpose of understanding the internals, as SSSD already implements universal responders for both NSS and PAM available for all of its providers. It is however useful to understand the inner workings to be able to identify a good solution we're going to build on top of it. More information about writing new NSS and PAM modules can be found in [11] and [22] respectively.

4.2 Talking to WinBind

As we've seen in the previous section about NSS and PAM, the interface between those system services and SSSD (or any other identity/authentication provider) is standardized and well defined. SSSD already has all the necessary hooks with the universal responders and lets us concentrate on developing specific responders. What's left for us to decide is how we're going to tackle communication with WinBind. We already know that WinBind has its own NSS and PAM responders, but they can't be just plugged into SSSD. In theory, it might be possible to export the appropriate calls from them and use SSSD as a mere proxy, but it would require some tricky linking and probably cause conflicts as SSSD already exports the same calls (by name). Therefore, we need to find another interface, that we can use to forward requests throught.

WinBind, like many other system daemons, has a named pipe it uses for communication with the outside world. On Fedora and many other distributions, this named pipe has a file, which can be found at `/var/run/winbind/pipe`. The problem is, that the communication protocol over this pipe is binary and isn't documented anywhere. It's only used internally by Samba. Even though Samba (the whole suite including WinBind) is open source software, it would still require a lot of reverse engineering to figure the protocol out and there's no guarantee it's not going to change in the future.

Fortunately, we're not bound to use this interface at all and it isn't recommended by the Samba team either. Instead of doing IO directly on the pipe, we can use a much friendlier API exported by the libwbclient library that comes with the WinBind client Samba package[1]. This API includes all the necessary calls for retrieving identity information

[1] It includes utility programs like wbinfo.

from NT and AD domain controllers and also for doing authentication against them, which is exactly what we need.

Using libwbclient as the only interface for requests and responses between SSSD and WinBind should be very confortable from the implementation point of view, because it's only a matter of linking with the correct static library. Required header files (or in this case only one file: `wbclient.h`) are packaged with it. We can also use the WinBind clients utility programs source code to figure out how to use the libwbclient API properly, which will be very helpful as documentation is sparse.

The only problem with this API is that it is completely synchronous. This effectively means that any request forwarded to WinBind using it will hang the provider process until results are returned. It's not a problem if WinBind doesn't have to query domain controllers for information to satisfy the request or if the piece of requested information is small and latency between the host system and the domain controller is low, but this is hardly the case in most scenarios. The API was designed for WinBind client utility programs such as wbinfo, which are very little concerned about performance. On the contrary in SSSD, we should be very concerned with performance, since our goal is for it to become the main gateway for identity and authentication for all NSS and PAM enabled applications. This work proposes to solve this problem by running a request handling process in parallel to the main provider process. They will communicate using an unnamed pipe. Requests will be stored in a FIFO buffer by the request handling process and satisfied one by one as they come in while the main process stays responsive to the environment. The solution is depicted in figure 4.2.

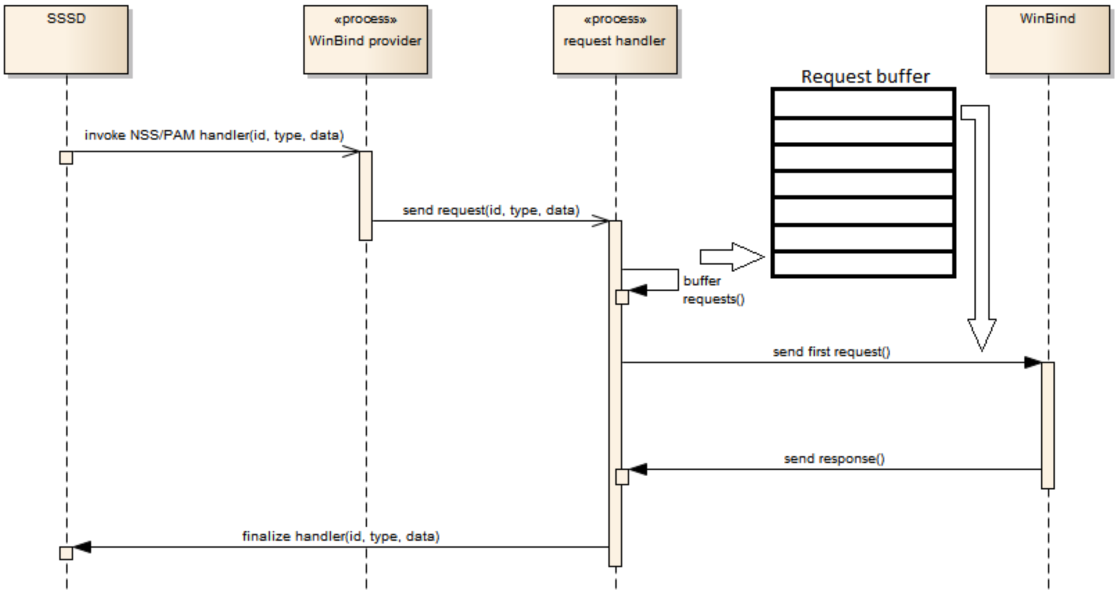

Figure 4.2: Communication between WinBind providers processes

It's out of the scope of this work, but it would be beneficial to create an asynchronous API on top of libwbclient, that could be used by other applications performance aware applications seeking information from WinBind. More about this in the last chapter about future work and extensions.

4.3 Controlling WinBind

After outlining the high level connections between NSS/PAM, SSSD and WinBind at the beginning of this chapter and defining the interface between the later two in the previous section, we're now standing before the decision of whether we want SSSD to take control of the WinBind process (in the sense of starting, restarting and stopping it) and to what extent. The answer for the first part of this decision is straigh forward. Taking control of the WinBind process is advantageous to us for the following reasons:

- Synchronization
- Unexpected WinBind behaviour (e.g. crashes)
- Configuration
- Multiple WinBind instances?

In this context, synchronization means that we want SSSD to start before WinBind and being able to know when WinBind is ready to answer request forwarded from NSS/PAM. We also want to be able to check if everything is well and works as expected. Both requirements should be easy to fulfill if we're in control of the daemons process. We won't discuss the configuration item in details at this point, as there is a whole section dedicated to it in this chapter. Let's just say, that its easier to handle WinBind's configuration if we can specify the parameters, which are passed to it on the command line at startup. As readers have probably noticed, there's a question mark at the end of the last item the list above this paragraph. This is not a coincidence or typo. With the current version of Samba and WinBind it is impossible to have more than one instance of WinBind running at the same time. It is therefore impossible to join the Linux hosts to more than one Active Directory domain tree[2]. Shortly after the project behind this thesis took off, negotiations with the Samba team for supporting multiple independent instances of WinBind have started. If these negotiations go well, it would be very advantagous to be able to spawn and kill WinBind process based on what domains are currently joinned on the host system. This feature would be very useful especially for laptop users who connect to the outside world through different domains at different locations (office, home, foreign office, etc.). Support for as many domains as possible at the same time without the need for reconfiguration is one of the main concern of SSSD.

Now that we know we want SSSD to take control of WinBind process(es) and we also know why, it's time to step a little further and look at the possibilities of how it can be achieved. There are three ways a daemon process can be controlled on Linux systems, that we can exploit for out purpose.

1. Using init scripts and/or related utilities (e.g. service)
2. Sending IPC[3] signals
3. Spawning the daemon process from whithin another the controlling process

[2] WinBind can only join one AD domain at the same time, but trusted domains are also supported effectively allowing access to the whole domain tree if not restricted by site-specific security policies.

[3] Inter-process communication

Init scipts and related utilities are here mostly for completness rather than being a real option. They are only standardized to a certain extent, each distribution out there handles them a little bit differently and we're not even considering other potential target platforms like Solaris and countless Unix flavors. `service` and similar utilities are also distribution specific (there's no single implementation). Furthermore they are primarily indented to be used manually by users from their shells.

On the contrary, sending signals is a valid option. They can be sent to any process running on the system if we have enough privileges to do so. The only requirement is that we need to know the process ID (PID). On Fedora and many other distributions, WinBind's PID can be retrieved from a special file with the .pid extention located in `/var/run/`. This file is named after the process, but in the case of WinBind it gets renamed if the configuration parameter (`-n`, discusses in section 4.4) is used. As a result of this, the file name is a little bit less predictable. It can also differ from one Linux distribution to another. Fortunately, we can always search the `/proc` file system to find out WinBind's PID portably. However, signals are only one component of the solution we're looking for. They give us the possibility to stop, kill and possibly (with the right signal handlers) restart the daemon we want to take control of, but they only work on already running processes. This means that we can't start the deamon if it wasn't started for us before hand by some external facility.

To take full control of the target deamon. We need to spawn its process ourselves from within the controlling process, which gets us to option three. In our case this means starting WinBind from the SSSD provider. This way, we can easily control when and with what parameters the process is started, we don't need to retrieve its PID for signal sending, because we know it first hand, and we have direct access to its return codes. Implementation details will be described in the next chapter.

As mentioned earlier, only one instance of WinBind can be running at any given time on the host system. Because of this limitation, we need to detect if it isn't running already when we try to spawn our own instance. It could be done by checking for existence of the PID file in `/var/run/` or searching the `/proc` filesystem as mentioned earlier, but there is another way easier than both of those. When investigating communication with the WinBind daemon in the previous section, we discussed the possibility of exploiting its named pipe. The name of this pipe file is always the same, therefore its easy to check for its existence for the purpose of determining if Winbind is running or not.

4.4 Configuration

Another problem that needs to be solved is how to handle configuration of Linux clients. We need to start from configuration is required by WinBind to succesfully connect and communicate with Windows NT and/or Active Directory domain controllers.

WinBind, like any other service provided by the Samba application suite, is configured using the `smb.conf` file. The format of this file is derived from the widely recognized (although never officially standardized) `INI` file format introduced by the Windows family of operating systems.

The `INI` file format defines properties identified by name and their value. In other words, it defines key value pairs delimited by equal signs. Each key value pair resides on a single line. Properties may be grouped into arbitrarily named sections. The start of a section is characterized by a its name enclosed insquare brackets on a new line by itself.

Samba recognizes several types of sections in its configuration file (`smb.conf`), but the majority of them is out of the scope of this thesis, as they are unuseful in the goal of joining Linux clients to Windows domain controllers. The only one we're interested in, is the `[global]` section. This is where all the properties related to WinBind functionality are located. Relevant properties with a short description of their meaning and acceptable values can be found in table 4.2.

Option	**Description**
workgroup	name of domain WinBind connects to
realm	Kerberos realm WinBind uses for authentication (can also be a KDC network address)
security	'domain' for NT domains and 'ads' for AD domains
password server	network address of domain controller to use
wins server	network address of WINS server if any
winbind enum users	Enables or disables user account enumeration.
winbind enum groups	Enables or disables user group enumeration.
winbind refresh tickets	Enables or disables automatic Kerberos ticket refreshing.
idmap backend	name of IDMAP backend to use (tdb, rid, ...)
idmap uid	range of UIDs NT/AD user account SIDs will be mapped to
idmap gid	range of GIDs NT/AD user account SIDs will be mapped to
template shell	default shell for NT/AD users logging in locally

Table 4.2: smb.conf options related to WinBind

Configuration of SSSD takes place in the `sssd.conf` file. It's format is similar to `smb.conf` at first sight, but acceptable keys and their values are defined more formally using a special grammar created specifically for this purpose.

Definitions of acceptable key value pairs are distributed with SSSD and are located in the `sssd.conf.d` subdirectory of where the main configuration file is located[4].

Since one of our goals defined in the start of this chapter was to make SSSD the only authority regarding identity and authentication of its host system, we don't want users to be required to modify the smb.conf file in order to configure WinBind. The only place where any configuration takes places should be the SSSD configuration file. To make this possible, we need to define new configuration options the SSSD WinBind provider, which map directly (or indirectly if it makes things more meaningful) to WinBind properties in `smb.conf`. This thesis proposes the mapping found in table 4.3. Notice that `winbind enum users` and `winbind enum groups` have been unified into the universal option `enumerate` that SSSD uses for all identity providers.

SSSD will read its own configuration and translate relevant parts to WinBind without the need of any interaction from users. This needs to happen on the fly, because the configuration can vary in time when the network topology or organizationl struture changes. Reading and translations are implementations problems, that will be analyzed in later chapters, but we need to figure out the best way of passing it to the WinBind daemon. We have basically three options:

1. Writing the translated configuration to the main `smb.conf`.

[4] /etc/sssd/ on most systems

SSSD option	**mapping to WinBind option**
winbind_workgroup	workgroup
winbind_realm	realm
winbind_domain_type	security
winbind_pwd_server	password server
winbind_wins_server	wins server
winbind_refresh_tickets	winbind refresh tickets
winbind_idmap_be	idmap backend
winbind_uid_min	low range limit of idmap uid
winbind_uid_max	high range limit of idmap uid
winbind_gid_min	low range limit of idmap gid
winbind_gid_max	high range limit of idmap gid
winbind_temp_shell	template shell
enumerate	winbind enum (users|groups)

Table 4.3: SSSD to WinBind option mapping

2. Writing a stand-alone `smb.conf` and passing it to the WinBind daemon using it's `-n` command-line option.

3. Using the Samba registry configuration.

Let's take a look at these options one by one. The first one might seem to be the most straight forward, but after analyzing the requirements for implementation, it comes out as the exact opposite. This is because, we can't just write whatever comes out of SSSD into the main Samba configuration file as we would be putting ourselves in the danger of overwritting parts related to other Samba services. We would also have to find the start of the `[global]` section and write underneath it. In the end, to make the whole process safe, we would need to parse the whole configuration file and rewrite it completely, which is an uneasy and error prone task. The second option, although similar to the first one, is much easier as we're only writting the relavant properties into a new blank file. We don't need to care about influencing any other Samba services, because the configuration file we're writting is isolated. As we're going to see in the last chapter, it is also advantageous if we ever want to have more than one instance of the WinBind daemon running at the same time. The last option makes use of the new configuration interface introduced in Samba version 3.2.0 - registry based configuration. This interface is derived from the Windows system-wide registry. While it is a lot more flexible (sections, properties and their values are hierarchically stored in a TDB[5] database) than flat files, it suffers from some of the same problems as using the main configuration file in the first option. It's not worth to go into details at this point, but it's important for readers to know about this possibility as it might be exploited in future extensions of Samba. Lookup [7] for more information about registry based configuration.

After analyzing all the relevant caveats of configuration, we can conclude that for our purpose, the best approach would be to: create new configuration options for SSSD and map them to WinBind properties normally found in the Samba configuration file (`smb.conf`). SSSD will read and translate them for WinBind on the fly into a new stand-alone configuration file, that will be passed to WinBind on the command-line using the `-s` option.

[5] Trivial database, pre-relational database engine developed by the Samba team [21].

Chapter 5

Implementing WinBind providers

The whole previous chapter was about the design of the WinBind providers for SSSD, but it was intentionally stripped of in depth technical details. In other words, it was more about what then how. In this chapter, we're going to go through what we planned and descibe how it was implemented, what tools have been used and what pitfall were encountered.

SSSD is an open source project with very strong emphasis on code culture. It's written completely in pure C89 without GNU extensions. Except for `talloc` and `tevent` (created originally for the Samba project), it only uses standard POSIX libraries with high portability in mind. For the WinBind provider, we need to introduce a new dependency, namely the `wbclient` library as discussed in section 4.2. In order to satisfy SSSD configure script, modifications to its standard issue header file `wbclient.h` are required, because it doesn't define several fixed size integer types (e.g. `int32_t`) and is therefore not „usable“ according to the script. The modifications in questions are simple. We only need to add a new `#include` directive for the standard C header `inttypes.h`. Since this way we get a custom version of the `wbclient` header file, we need to ship it with SSSD to override the system wide include.

We also need to make SSSD aware of the new configuration options related to the WinBind provider we defined in section 4.4. Configuration options are interpreted by providers freely, but they can't be arbitrary in the sense, that they need to be predefined in files localed in the `sssd.conf.d` directory. The format of these files is as follows:

```
option = <type>, <subtype>, <mandatory>[, <default>]
```

Where `<type>` is either `str`, `int` or `list` according to what type it should be read as by SSSD configuration API. `<subtype>` is only useful if `<type>` is set to `list`. `<mandatory>` is either `true` or `false` and doesn't need further explanation. `<default>` is optional and can be anything that can be interpreted as a valid value of the corresponding type. Actual definitions of options introduced in 4.4 are listed in appendix C.

5.1 SSSD backend framework

SSSD has a built-in framework for creating new backends. The term backend in this context is just an abstraction for a set of up to four providers each responsible for a specific function. The four provider types are characterized in table 5.1. In fact, any provider type can exist independently by itself and doesn't have to be tied to the other types in any way.

Provider	Description
ID	Identity used to handle NSS requests
AUTH	Authentication used to handle PAM auth requests
ACCESS	Security policy used to handle PAM account requests
CHPASS	Changing authentication tokens used to handle PAM chpass requests ...

Table 5.1: SSSD provider types

It's a common practice in pluggable architectures to have a registration mechanism in place for new plugins. In SSSD, however, this isn't the case. New providers don't have be registred or listed anywhere and are detected automatically by looking up the expected initialization function according to the SSSD configuration file. The monitor process, that keeps all providers well and running, is responsible for this task. For the appropriate initialization function to be found by the monitor, it needs to have the following signature:

```
int sssm_<backend-name>_<provider-type>_init(struct be_ctx *, struct bet_ops **, void **)
```

Where `<backend-name>` is the name of the backend this provider will be part of and `<provider-type>` is one of the four types found in table 5.1. Both fields need to be lowercase.

The initialization functions take three arguments out of which only the first one serves as input and the other two are to be filled in when the function returns. The type `struct be_ctx` of the first argument represents the backend context. It contains about twenty different attributes, but most of them are used internally by the framework. Following is a list of the most important attribute, we're going to use when implementing the WinBind providers:

- `struct confdb_ctx *cdb` - configuration database context
- `struct sysdb_ctx *sysdb` - system database context
- `struct sss_domain_info *domain` - domain information

Configuration and system database contexts are very important variables used to access options from the configuration file in case of the first mentioned and cached identities and authentication tokens in case of the second. We're going to see them in action later in this chapter. Domain information, as the name suggests, contains information about the domain this provider belongs to.

The second argument is supposed to be filled in with a pointer to a `struct bet_ops` structure. This structure contains function pointers to up to three callbacks:

- handler is the main callback for operations handled by the provider.
- check_online is used to check if the backend is online.
- finalize is called when SSSD shutdowns.

handler is the most important callback and should be always present. The other two are optional. Checking if the backend is online or offline is useful when the source of identity, authentication or policy information is queried remotely over the network, which is the case of our WinBind provider, because this information is retrieved from NT/AD domain controllers. When SSSD shutdowns, the monitor process triggers the finalize callback on all providers to give them the opportunity to free used resources and perform other necessary clean up procedures. At least, that's how it's supposed to be, because at the time of this writing, finalize is never triggered and providers have to implement their own mechanisms for cleaning up.

The last argument is used to store private data shared between provider callbacks.

5.2 Initialization

Now that we've been throught the basics of the backend framework, it's time to dwelve into the specific details of the implemented WinBind providers. Each provider needs to be initialized separately, but the majority of steps is shared between them. These steps are ensued from what has been planned in chapter 4 and are as follows:

1. Initializing backend private data
2. Retrieving configuration options
3. Creating a dynamic WinBind configuration file
4. Spawning the WinBind daemon

In step one, we need to initialize a new structure to hold key pieces of information required for handlers of all the implemented providers. This structure is named struct winbind_id_ctx according to the customs introduced by other backend modules. We won't show its whole definition here, but it's important to know that we're going to use it to store the WinBind daemon process ID (PID) and options from SSSD configuration file related to our backend. The whole definition can be found in appendix D.

Step two, on the contrary of what might be expected, doesn't involve any IO operations, because the backend framework has already read all configurations options for us according to what was specified in sssd.conf.d (as described at beginning of this chapter). All that's left for us to do is to retrieve the relevant values, perform validation and store them in the private data structure from step one for later use. Retrieving the values is done using functions defined in providers/dp_backend.h such as dp_get_options, dp_opt_get_string, dp_opt_get_integer, . . . The first mentioned is used to retrieve all the relevant values in bulk and needs a statically defined array of struct dp_option structures. This array has to correspond to what is defined in sssd.conf.d. Its definition for our WinBind providers is displayed in Appendix C. All other of the mentioned functions are used to retrieve single values and are usuful to us for validation of the winbind_domain_type option. This step is implemented in providers/winbind/winbind_common.h as the function winbind_get_options called directly from the ID provider initialization function.

Another implementation option for step two would be to use `libsmbconf` as discussed in section 4.4, but a desicison against it was made for two main practical reasons implied by the fact that the library is still in developement and unstable. The first reason is that it isn't available anywhere else than in a non-master branch of the Samba git repository, causing packaging and distribution problems for the final product. The second reason being that the library compiles with uncorrectly hard-coded paths of crucial Samba files and directories (such as the path to TDB files).

The third step involves writting a configuration file for WinBind. This is performed on the fly from what has been retrieved in the previous step according to table 4.3 in function `winbind_write_config`. The file is truncated everytime to ensure that there's no leftovers from another run. By default, it's stored as `/var/lib/sssd_winbind.conf` and this path is later passed to the WinBind daemon using its `-n` option. Except for `enumerate` and UID/GID ranges, the whole process is implemented by simply using a static conversion table.

The last step is the most important and complex and therefore deserves its own dedicated section of this paper.

5.3 Spawning WinBind

A common technique of spawning a new program from the executing process on POSIX operating systems is the so called *fork-exec*. The technique takes its name from two system calls it makes use of. It's probably not a surprise, that the first of these calls is `fork`. It effectively creates a new process identical to the the one it was forked from. The parent process gets the childs PID in the return value and we're going to store in the `struct winbind_id_ctx` structure, because this will soon become the process ID of the WinBind daemon. In the child process, we're going to use the techniques second system call - `exec` or more precisely one of its variants - `execvp`. The `exec` family of functions overrides the calling process with a new image created from a file descriptior or, in the case of the variant we're going to use, from a file specified with a path string. They also allow us to pass parameters to the new image.

However, before we start spawning anything, we need to make some preparations. It was noted in section 4.3, that currently only one instance of the WinBind daemon can be running at any given time. Even if this state is temporary, as of now we need to check if the daemon is running or not. Checking it is implemented by using the `stat` system call on the WinBind daemon named pipe file. If the call fails and `errno` is set to `ENOENT`[1] we can safely continue. Otherwise, we're going to display an error message and stop the WinBind provider initialization process. It would be possible to stop the currently running daemon process, but it was decided not to do so prevent possible configuration breakdown as there's no way for us to know why it was started in the first place.

Before forking and once the necessary check for running instance of the WinBind daemon, we should also make sure that all opened file descriptors have the `CLOEXEC` flag set. This flag ensures that descriptors are closed when `exec` is called. It's never set by default, because it's a common practice to open an unnamed pipe just before using the *fork-exec* technique, so that the parent and child can communicate throught it. Fortunately, the implemented WinBind providers shouldn't have any opened file descriptors at this point

[1] A component of the path does not exist, or the path is an empty string.

except for those started by the monitor process and these are already set with the appropriate flags.

When everything is setup, we can finally fork the running process and override its child with the WinBind daemon program. It should be accessible using the PATH environment variable if the installation of Samba is configured correctly on the host system as winbindd. The overridden process will receive the following parameters:

```
winbindd -F -n -s /var/lib/sssd_winbind.conf
```

Where the -F option tells WinBind to run in *foreground mode*, which means that it's not going to deamonize, i.e. double fork and disassociate with the terminal [19]. In our case, role of the terminal is taken by the child process calling exec. We don't want it to deamonize, because its main process PID would change from what we got from fork and it would be harder to kill when SSSD shutdowns as we're going to see later in this section. The -s option is used to specify the configuration file we generated in winbind_write_config.

Notice that the -n option was intentionally omitted from the description of parameters in the previous paragraph. It has a special and important meaning. According to [19] it should disable WinBind caching mechanism, which is one of the main goals we set for ourselves at the beggining of chapter 4. However, after examining the daemon source code carefully, the conclusion that it's not entirely true was reached. At the time of this writting, negotiations with the Samba team are in progress regarding this issue.

Even after the WinBind daemon is successfully spawned, we're not done yet. Signal handlers have to be setup in the parent process specifically SIGTERM and SIGCHLD. The first one needs to be handled, because we can't rely on the finalize handler as discussed in the previous section and we need to get a chance to kill the daemon when SSSD shutdowns. Killing it is straight forward; it involves nothing more that sending the SIGTERM signal to the previously stored PID in struct winbind_id_ctx. The implementation handles SIGCHLD to detect when the daemon stops running for some reason. If this happens, we can safely get its return code using the system call waitpid and respawn it if necessary.

All of the tasks discussed in this section (with the exception of signal handlers) are performed by the winbind_start_daemon function implemented in providers/winbind/winbind_common.c.

5.4 ID provider

This providers initialization function is called sssm_winbind_id_init according to conventions discussed in the first section of this chapter. You can find it in providers/winbind/winbind_init.c. It performs all of the required preparations such as reading configuration, generating files for the WinBind daemon and spawning it. It also initializes the private data structure struct winbind_id_ctx and fills all of its fields. Opposite to what its name would suggest, it isn't specific to the ID provider and is shared among all providers of the implemented backend.

As noted in table 5.1, the main responsibility of ID providers is to handle NSS requests that can't be satisfied from SSSD identity cache. winbind_account_info_handler implemented in providers/winbind/winbind_id.c is the primary callback for handling such requests. This callback serves as a router for different types of queries and is implemented as a group of nested switches. Identifying the type of NSS request and choosing the right query is based determined using the `req_data` field of the `struct be_req` structure that comes in as the only parameter to the primary handler. This structure is universal to all providers and must be cast to `struct be_acct_req` in the case of ID. From there we can retrieve what entry type is looked up and with what kind of filter. Both are represented by constant defined in `providers/data_provider.h`. An overview of possible entry type and filter constants is shown in table 5.2 and table 5.3 respectively.

Constant	Description
BE_REQ_USER	User accounts
BE_REQ_GROUP	User groups
BE_REQ_INITGROUPS	Groups a specific user is member of BSD introduced concept for group access lists
BE_REQ_NETGROUP	Netgroups NIS introduces concept of groups containing hosts and users

Table 5.2: NSS entry type constants

Constant	Description
BE_FILTER_NAME	Search by name
BE_FILTER_IDNUM	Search by ID (UID/GID/...)
BE_FILTER_ENUM	Enumerate all

Table 5.3: NSS filter type constants

An important aspect about ID providers is that they don't return anything. There's no response to requests coming from NSS. Instead these providers just update the internal cache called *sysdb* with identity information. The implemented ID provider is no exception to this rule.

Requests tagged with the `BE_FILTER_NAME` and `BE_FILTER_IDNUM` constants combined with `BE_REQ_USER` are routed to the `winbind_get_user` function. This functions retrieves the requested identity information from WinBind and stores it using the `sysdb` API, which provides functions with parameters mapping to fields in the POSIX defined `struct passwd` which is very handy, because that's exactly what is returned by the `wbclient` library. Same goes for these constants combined with `BE_REQ_GROUP`, except that it's routed to `winbind_get_group` and the prominent POSIX structure is `struct group`.

Any request with the last filter type is only satisfied if emuration for the domain is enabled in `sssd.conf`. The query is executed by `winbind_enum_users` or by `winbind_enum_groups` according to the entry type. One specialty of enumeration is that more than one entry is updated in the cache and the whole operation is therefore non-atomic. If interrupted for some reasons, it might cause inconsistent states, which is something we need to avoid. Situations like this were thought of when `sysdb` was designed and the support for transactions is available. It only requires programmers to call `sysdb_transaction_start` before updating anything and `sysdb_transaction_commit` when complete.

Our ID provider also implements the `check_online` callback. Its operation is simple thanks to the fact, that the wbclient library provides an API call tailored for retrieving information about NT/AD domains. This information contains various domain flags among which WBC_DOMINFO_DOMAIN_OFFLINE is what we're looking for. If present, we know for sure the domain is offline.

5.5 AUTH provider

The authentication provider relies on the ID provider for performing the required initialization steps. To be registred by the SSSD monitor daemon it needs a standalone initialization function named sssm_winbind_auth_init. Since the order of provider registration is not guaranteed, it start by calling the ID provider initializer that contains checks if it has been already called or not, so that configuration isn't read needlessly twice and more importantly, so that we're not trying to spawn the WinBind daemon more than once.

AUTH providers are responsible for satisfying PAM requests as noted in table 5.1. The main callback here is called winbind_pam_auth_handler and can be found in providers/winbind/winbind_auth.c. Similarly to the main ID provider handler, it also serves as a router for different types of queries and is implemented using switches, although this time there's no nesting involved. There is only one switch for PAM operations. These operations correspond closely to PAM service module functions. These functions are implemented by responder modules to handle requests and SSSD is no exception. In detail description of these functions is out of the scope of this thesis, for more information see [16]. Individual operations are identified on the basis of the req_data field of the be_req structure. Note that the handler callback for AUTH providers has the same signature as for ID providers. The difference is that the prominent field needs to be cast to the struct pam_data structure. A field containing one of the constants in table 5.4 can be dereferenced from it after the cast.

Constant	**Description**
SSS_PAM_AUTHENTICATE	Authenticate user
SSS_PAM_SETCRED	Alter user credentials
SSS_PAM_ACCT_MGMT	Decide if user has access
SSS_PAM_OPEN_SESSION	Commence a session
SSS_PAM_CLOSE	Terminate a session
SSS_PAM_CHAUTHTOK	(Re)set authentication token 2
SSS_PAM_CHAUTHTOK_PRELIM	(Re)set authentication token 1
SSS_CMD_RENEW	Refresh credentials with limited lifetime

Table 5.4: PAM operation constants

The most important PAM operation handled by the AUTH provider is marked by the constant SSS_PAM_AUTHENTICATE and is triggered when a user tries to login using his password. The main handler routes this to the winbind_pam_authenticate function. This function queries the domain controller and tries to authenticate the user with the provided credentials. The password is sent to WinBind in clear text form, but it doesn't present a security risk because the daemon encrypts it before it leaves the host system. If success is reported, the credentials are cached along with the user entry in sysdb to allow offline authentication.

A major difference between ID and AUTH providers is that the later must return a value representing the end state of the requested operation. The set of valid values depends on what was requested. Possible return values for SSS_PAM_AUTENTICATE are displayed in table 5.5. winbind_pam_authenticate translates whatever is returned by WinBind to one of these values except for PAM_CRED_INSUFFICIENT, because it doesn't make sense for passwords.

Constant	**Description**
PAM_AUTH_ERR	Authentication failure
PAM_CRED_INSUFFICIENT	Not enought credentials
PAM_AUTHINFO_UNAVAIL	Unable to access authentication information
PAM_SUCCESS	Authentication success
PAM_USER_UNKNOWN	Username provided is invalid
PAM_MAXTRIES	Maximum number of authentication tries was reached

Table 5.5: PAM Authenticate return values

SSS_PAM_CHAUTHTOK and SSS_PAM_CHAUTHTOK_PRELIM deserve some additional explanation. They're both related to a single service module function - pam_sm_chauthtok. This function is called when authentication tokens for a specific user are about to be changed. It's called twice. Once to check if and a second time to actually change the tokens [16]. The first call is equivalent to the prelimitary operation and is the only one Our AUTH provider is going to handle as it's equivalent to SSS_PAM_AUTHENTICATE for NT/AD password. Handling the second call if the responsibility of the CHPASS provider.

Remaining operations are unimplemented, because they are uninteresting from network authentication point of view. The statement isn't true about SSS_CMD_RENEW. This operation is useful for authentication tokens with limited lifetimes such as Kerberos tickets used by some AD services. However we don't need to take care of them as WinBind does it automatically given the correct settings.

check_online isn't implemented as part of the AUTH provider, because it's already handled by ID provider, which sets the online/offline status for the whole backend.

Chapter 6

Testing and evaluation

In the final chapter of this thesis, we're going to go through the procedures that have been continuously used to test the implementation outputs during the whole developement process. We're also going to evaluate what has been achieved and where this project is going in the future. Due to its nature, there are no numbers, nice graphs with performance curves. Instead we're going to present the area where the outputs are useful in real life situations.

To take full advantage of SSSD and the implemented WinBind backend a single machine is not enough. We need connectivity to a domain controller be it NT or AD. The first section of this chapter will guide readers through the basic steps of setting up such an environment where the implementation can be tested for the most part.

6.1 Environment preparations

Due to the fact that SSSD is developed by Red Hat, it's native distribution are the in-house distribution of Linux: Fedora and Red Hat Enterprise Linux. Not that it wouldn't run on other distributions as well, but we still recommend you to use one of them. For the purpose of this work, we're going to pick Fedora as it's available for free and more accessible to a wider audience. We're going to use it as the host system for SSSD and a client of the an Active Directory domain.

We're going to assume that most readers don't have access to a readily available AD domain for testing purposes. Therefore we need to create one. All we need is one domain controller. Samba can act like one, but everything in this chapter has been tested against an 'original' Active Directory enabled server running Windows 2008 R2 Server. For reference about configuring AD on this operating system consult [18].

Both system (the host for SSSD and the domain controller) must have network access to other preferrably on a private network. Ports required by AD must be open and you might consider disabling the firewall on both machines if your testing environment is isolated.

The host system needs to join the Active Directory domain as a member. This can be achieve either manually using the Samba provided `net` utility:

```
net ads join -w DOMAIN -I SERVERIP -U USERNAME%PASSWORD
```

Or by using the `join-ad.sh` script on the CD that comes with this paper:

```
./join-ad.sh DOMAIN SERVERIP USERNAME PASSWORD
```

The later is recommended, because it not only joins the domain, but also checks if all the required packages are installed on the host system.

6.2 Testing procedures

This section focuses on a few selected use cases and simple procedures tailored to demonstrate the implemented functionality.

Configuration

The first testing procedure is about checking that if configured corretly, the SSSD monitor process starts the WinBind backend and both of it's implemented providers.

1. Add an a new accessible AD domain to `sssd.conf` using WinBind providers
2. Start SSSD: `sssd -d 3 -i`
3. Analyze the standard error output
4. Check that WinBind is running: `service winbind status`
5. Check the generated WinBind configuration file

The options used in step two make SSSD start in `interactive mode` with debug level three. This effectively means that it will not daemonize and run in the foreground with debugging messages up to level three showing up on standard error. Without these options, we wouldn't be able to analyze the output and see what providers have been initialized. If the WinBind daemon was already running an error message should be reported.

An example of a valid `sssd.conf` and the generated configuration file for WinBind used when writing the testing procedures is on display in appendix C.

Looking up users and groups

This procedures is for checking that looking up identity information for a single entity works as expected. Successful completion of the previous procedure is a required prerequisite.

1. Make sure SSSD and WinBind are running: `service sssd status`
2. Create a new user in AD with name 'testuser'
3. Retrieve the newly created user on the host system: `getent passwd DOMAIN\testuser`
4. Create a new group in AD with name 'testgroup'
5. Retrieve the newly created group on the host system: `getent group DOMAIN\testgroup`

`getent` is the standard NSS enabled utility for retrieving entries from identity repositories. `passwd` ang `group` are aliases for available user account and user group databases.

If everything works as expected, commands from step three and five should display information about the newly created entries in AD and should have an UID/GID assigned to them. Retrieving an entry that doesn't exist should leave the commands with no output.

Enumerating users and groups

Enumerating users and groups is very similar to the previous procedure except that this time, we're retrieving all available entries at once. This includes all available entries and not just the ones in the AD domain. `enumerate` must be enabled in `sssd.conf`.

1. Make sure SSSD and WinBind are running: `service sssd status`
2. Retrieve all users available on the host system: `getent passwd`
3. Retrieve all groups available on the host system: `getent group`

Output of the `getent` commands should include entries from the AD domain. The listings can be compared to the output of `wbinfo -u` and `wbinfo -g` to make sure that all entries are really shown. If `enumerate` is disabled, only users and groups from the local domain (e.g. from `/etc/passwd`) should be displayed.

Authenticating as a domain user

The last testing procedure involves authenticating as an AD domain user locally on the host system. The user we're going to authenticate as must have a valid unlocked account in the AD domain and a password set. Another important requirement is that the `winbind_temp_shell` option in `sssd.conf` must be set to an existing shell on the host system such as bash or ksh.

1. Make sure SSSD and WinBind are running: `service sssd status`
2. Create a new user in AD with name 'testauthuser'
3. Login as the newly created user on the host system: `su - DOMAIN\testauthuser`
4. Check the user is logged in: `id`
5. Try to create files and analyze the owner and group they belong to

If everything is configured correctly, the current working directory should be changed to the AD user home directory. This directory should be created automatically on first login. Files created by the domain user should have owner set to him and group set to `DOMAIN\Domain users`.

6.3 Future roadmap

This section discusses some of the planned extension for the future of the whole project and the motivation behind them. There's still a lot of work to be done for the project to become a widely adopted solution for the authentication of Linux clients in Windows domains. What has been implemented at this point is just the beginning - a seed of a much larger scheme.

Additional providers

Currently, only the ID and AUTH providers of the WinBind backend have been implemented, which means that tasks like changing authentication tokens and enforcing security policies defined in NT/AD directory services is not possible. A CHPASS provider needs to be implemented and the `wbclient` library has means to make it conceivable. When it comes to security policies, future plans are unclear at the time of this writing. WinBind doesn't have any facilities to enforce or even display any of it. Extentsion to the daemon or another idependent solution will be necessary.

The AUTH provider also need to have more fine grained options and Kerberos possibilities of Active Directory have yet to be fully exploited.

Cross-domain trust

Windows domains can have trust relationships setup between each other. This means that services provided by one domains can be accessible to users from another one and vice versa. Cross-domain trust is an important aspect when building complex network infrastructures. WinBind by itself supports this feature, but SSSD doesn't.

We will need to implemented support for nested domains in SSSD. Large scale changes ranging from configuration to the backend framework are going to be necessary. Trust relationships of Windows domains can be created and removed on the fly at any given moment and dynamic detection of new domains in the tree has to be implemented.

Multiple domains

As noted in section 4.3, it's currently impossible to have more than one instance of WinBind running at once. Therefore Linux clients using it can become members of only one Windows domain at the same time. This is an unconfortable limitation especially for user authenticating through different domains on a daily basis (e.g. laptop users on business travels). The only way to solve this is to add the possibility of multiple WinBind instances. Negotiations with the Samba team have started and look promising.

Asynchronous API for WinBind

The current version of the WinBind API implemented by the `wbclient` library provides only synchronous interfaces, which is a major disadvantage especially when dealing with geographically spread networks. Even thought it's mitigated to a great extent by SSSD caching mechanism, it's still not insignificant.

There are two possible solutions to this problem. One is to update the WinBind API with asynchronous calls, which would require the collaboration of the Samba team, but could benefit other projects as well. The second one is to create an asynchronous layer on top of it for SSSD. This layer could be implemented specifically to our needs and be fitted directly into the `tevent` mechanism already in place.

Links to FreeIPA project

The FreeIPA project discussed in section 2.4 is running in parallel with SSSD under the same leadership. Long term goals include the possibility to create cross-domain trust relationships between Windows and IPA domains. The plan is to build the trust using the WinBind backend implemented as part of this work.

Chapter 7

Conclusion

The main focus of this thesis was the network authentications of Linux client against different directory services with the goal to design and implement a unified network authentication solution by integrating the WinBind and SSSD system daemons. The later provides a generic interface above NSS/PAM for identity and authentication services over the network. By integrating the two daemons, we get more control over available user and group domains, support for offline authentication, easier and portable configuration and much more.

According to the given project instructions, this work begins with investigation of the current state of the art in network authentication solutions in both Linux and Windows worlds. It starts with legacy technologies like NIS and ends with the latest projects in this area. A lot of attention was given to designing a robust approach to integrating WinBind as a new provider into the pluggable backend framework of SSSD. Possibilities were discussed and decisions made. The following part of the thesis takes it from there and goes through the technical details and pitfalls associated with implementing the proposed solution. Final pages are devoted to providing comprehensive instructions to reproduce test performed on the final product and giving an overview of the projects future roadmap.

Both formal and personal goals behind this work were met with success as the resulting WinBind provider for SSSD is fully usable. However, there is still much to be done. What has been implemented at this point is just the beginning - a seed of a much larger scheme for Active Directory integration.

Appendix A

List of abbreviations

ADDS Active Directory Domain Services

AD Active Directory

API Application Programming Interface

AS Authentication Server

BDC Backup Domain Controller

D-H Diffie-Hellman

DC Domain Controller

DIT Directory Information Tree

DNS Domain Name Service

GID Group ID

GSS Generic Security Services

KDC Key Distribution Server

LDAP Lightweight Directory Access Protocol

MIT Massachusetts Institute of Technology

MPI Message Passing Interface

MS Microsoft

NFS Network File System

NIS Network Information Service

NSS Name Service Switch

NTDS NT Directory Services

NTLM NT LAN Manager

NTP Network Time Protocol

NT Windows NT family of operating systems

PAM Pluggable Authentication Modules

PDC Primary Domain Controller

PID Process ID

RPC Remote Procedure Call

SASL Simple Authentication and Security Layer

SFU Windows Services For Unix

SID Security ID

SSL Secure Socket Layer

SSSD System Security Services Daemon

TDB Trivial Database

UID User ID

Appendix B

Comparison of Directory Services

	NIS	NIS+	LDAP	FreeIPA
Target Platform	POSIX	POSIX	-	Linux
Hierarchical directory	×	√	√	√
Complex Data	×	×	√	√
Replication	×	√	√	√
Multi-Master Replication	×	×	×/√	√
Security Protocols	D-H	D-H	SSL, SASL/MD5	SSL, Kerberos
Communication Protocol	Sun RPC	Sun RPC	LDAP v2, v3	LDAP v2, v3, IPA XML-RPC, IPA JSON-RPC

Table B.1: Comparison of Linux Directory Services

	NTDS	ADDS
Target Platform	Windows	Windows
Hierarchical directory	√	√
Complex Data	×	√
Replication	√	√
Multi-Master Replication	×	√
Security Protocols	NTLM	Kerberos, NTLM
Communication Protocol	MS RPC	MS MPI, LDAP v2, v3

Table B.2: Comparison of Windows Directory Services

Appendix C

Configuration samples

WinBind provider option definitions

```
[provider/winbind]
winbind_workgroup = str, None, false
winbind_realm = str, None, false
winbind_domain_type = str, None, false
winbind_pwd_server = str, None, false
winbind_wins_server = str, None, false
winbind_refresh_tickets = bool, None, false
winbind_idmap_be = str, None, false
winbind_uid_min = int, None, false
winbind_uid_max = int, None, false
winbind_gid_min = int, None, false
winbind_gid_max = int, None, false
winbind_temp_shell = str, None, false
```

Example `sssd.conf`

```
[domain/REDHAT]
description = WinBind integration
id_provider = winbind
enumerate = true
winbind_workgroup = REDHAT
winbind_realm = REDHAT.RH
winbind_domain_type = ads
winbind_pwd_server = w2k8server.redhat.rh
winbind_wins_server = w2k8server.redhat.rh
winbind_idmap_be = rid
winbind_refresh_tickets = true
winbind_uid_min = 1000000
winbind_uid_max = 2000000
winbind_gid_min = 1000000
winbind_gid_max = 2000000
winbind_temp_shell = /bin/bash
```

Example sssd_winbind.conf

```
[global]
workgroup = REDHAT
realm = REDHAT.RH
security = ads
password server = w2k8server.redhat.rh
wins server = w2k8server.redhat.rh
idmap backend = rid
winbind refresh tickets = Yes
template shell = /bin/bash
winbind enum users = Yes
winbind enum groups = Yes
idmap uid = 1000000-2000000
idmap gid = 1000000-2000000
```

Appendix D

Code samples

sample from winbind_common.h

```
struct winbind_id_ctx {
   struct be_ctx *be;
   int entry_cache_timeout;
   struct dp_options *opts;
   pid_t winbind_process_pid;
};
```

sample from winbind_common.c

```
#define WINBIND_CONF_FILE ,,/var/lib/sssd_winbind.conf‘‘
#define WINBIND_PIPE_FILE ,,/var/run/winbindd/pipe‘‘
const char * const winbind_daemon_argv[] = {
   ,,winbindd‘‘, ,,-F‘‘, ,,-n‘‘, ,,-s‘‘, WINBIND_CONF_FILE, NULL
};


struct dp_option default_winbind_opts[] = {
   { ,,winbind_workgroup‘‘, DP_OPT_STRING, NULL_STRING, NULL_STRING },
   { ,,winbind_realm‘‘, DP_OPT_STRING, NULL_STRING, NULL_STRING },
   { ,,winbind_domain_type‘‘, DP_OPT_STRING, NULL_STRING, NULL_STRING },
   { ,,winbind_pwd_server‘‘, DP_OPT_STRING, NULL_STRING, NULL_STRING },
   { ,,winbind_wins_server‘‘, DP_OPT_STRING, NULL_STRING, NULL_STRING },
   { ,,winbind_idmap_be‘‘, DP_OPT_STRING, NULL_STRING, NULL_STRING },
   { ,,winbind_refresh_tickets‘‘, DP_OPT_BOOL, BOOL_FALSE, BOOL_FALSE },
   { ,,winbind_uid_min‘‘, DP_OPT_NUMBER, NULL_NUMBER, NULL_NUMBER },
   { ,,winbind_uid_max‘‘, DP_OPT_NUMBER, NULL_NUMBER, NULL_NUMBER },
   { ,,winbind_gid_min‘‘, DP_OPT_NUMBER, NULL_NUMBER, NULL_NUMBER },
   { ,,winbind_gid_max‘‘, DP_OPT_NUMBER, NULL_NUMBER, NULL_NUMBER },
   { ,,winbind_temp_shell‘‘, DP_OPT_STRING, NULL_STRING, NULL_STRING },
};
```

Appendix E

CD contents

`sssd/`
GIT snapshot of SSSD branch `pzuna-winbind`

`0001-Winbind-provider-initial-commit.patch`
Patch for SSSD `master` branch

`ads-join.sh`
Utility script for joining Linux hosts to AD domains

`xzunap00.pdf`
Thesis text in PDF format

Bibliography

[1] LDAP Authentication Using pam_ldap and nss_ldap. http://www.saas.nsw.edu.au/solutions/ldap-auth-pam.html. (visited on January 2011).

[2] NIS+ End-of-Feature (EOF) Announcement FAQ. http://www.sun.com/software/solaris/faqs/nisplus.xml. (visited in January 2011).

[3] NIS+ to LDAP Migration in the Solaris(tm) Operating Environment. http://www.sun.com/software/whitepapers/solaris9/nisldap.pdf. (visited in January 2011).

[4] NT LAN Manager (NTLM) Authentication Protocol Specification. http://msdn.microsoft.com/en-us/library/cc236701%28v=PROT.10%29.aspx. (visited on January 2011).

[5] The GNU C Library Reference Manual. http://netbsd.gw.com/cgi-bin/man-cgi?nsswitch.conf+5+NetBSD-current. (visited on January 2011).

[6] Windows Services for Unix. http://technet.microsoft.com/en-us/library/bb496506.aspx. (visited on January 2011).

[7] Michael Adam. Samba's New Registry Based Configuration. http://www.samba.org/~obnox/presentations/linux-kongress-2008/lk2008-obnox.pdf. (visited on May 2011).

[8] Russ Allbery. pam-krb5. http://www.eyrie.org/~eagle/software/pam-krb5/. (visited on January 2011).

[9] Beekmans G. *Linux From Scratch*. IUniverse.com, Inc., 2000. ISBN 0-595-13765-2.

[10] Ray W. Hiltbrand. NIS+ FAQ. http://www.eng.auburn.edu/users/rayh/solaris/NIS+_FAQ.html. (visited in January 2011).

[11] The SCO Group. Inc. NSS Overview. http://uw714doc.sco.com/en/SEC_admin/nssover.html. (visited on May 2011).

[12] David Collier-Brown Jay Ts, Robert Eckstein. *Using Samba, 2nd Edition*. O'Reilly & Associates, 2003. ISBN 0-596-00256-4.

[13] Gerald Carter Jelmer Vernooij, John Terpstra. The Official Samba 3.5.x HOWTO and Reference Guide. http://samba.org/samba/docs/man/Samba-HOWTO-Collection/. (visited on January 2011).

[14] Theodore Y. T'so John T. Kohl, Clifford Neuman. *The Evolution of the Kerberos Authentication System*. IEEE Computer Society Press, 1994. ISBN 0-8186-4292-0.

[15] Luke Mewburn. Name Service Switch Configuration File FreeBSD Manual Page. http://netbsd.gw.com/cgi-bin/man-cgi?nsswitch.conf+5+NetBSD-current. (visited on January 2011).

[16] Andrew G. Morgan. The Linux-PAM Module Writers' Guide. http://www.kernel.org/pub/linux/libs/pam/Linux-PAM-html/old/pam_modules.html. (visited on May 2011).

[17] Tony Northrup. *Introducing Microsoft Windows 2000 Server*. Microsoft Press, 1999. ISBN 1-57231-875-9.

[18] John Policelli. *Active Directory Domain Services 2008 How-To*. Sams Publishing, 2009. ISBN 0-672-33045-8.

[19] Tim Potter. winbindd Manual page. http://www.manpagez.com/man/8/winbindd/. (visited on May 2011).

[20] J. Sermersheim. RFC 4511 - LDAP: The Protocol. http://tools.ietf.org/html/rfc4510. (visited in January 2011).

[21] Samba team. tdb Documentation. http://tdb.samba.org. (visited on May 2011).

[22] Jennifer Vesperman. Writing PAM Modules. http://linuxdevcenter.com/pub/a/linux/2002/05/02/pam_modules.html. (visited on May 2011).

[23] Kurt D. Zeilenga. RFC 4510 - LDAP: Technical Specification Roadmap. http://tools.ietf.org/html/rfc4510. (visited in January 2011).