

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SUPPORT FOR RADIUS PROTOCOL IN SSSD

BAKALÁŘSKÁ PRÁCE

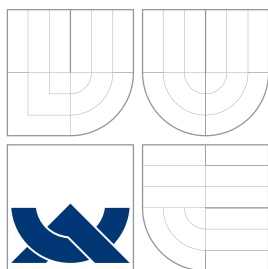
BACHELOR'S THESIS

AUTOR PRÁCE

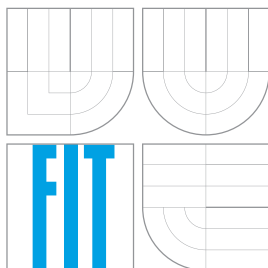
AUTHOR

ONDŘEJ HUJŇÁK

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PODPORA RADIUS PROTOKOLU V SSSD

SUPPORT FOR RADIUS PROTOCOL IN SSSD

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

ONDŘEJ HUJŇÁK

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. JAN ZELENÝ

BRNO 2013

Abstrakt

Moderní trendy ve správě uživatelů ve firemních prostředích směřují k centralizovaným řešením jako je LDAP či Active Directory. Ověřování uživatelů vůči těmto úložištím v Unix-like systémech je dostupné buď přes PAM moduly, nebo nově i přes bezpečnostní démon SSSD. Tato práce analyzuje využití RADIUS protokolu pro ověřování uživatelů a v rámci práce byl vyvinut modul do SSSD umožňující využití tohoto protokolu.

Abstract

Modern trends in user management in enterprise solutions makes use of centralized solutions such as LDAP or Active Directory. User validation against those resources in Unix-like systems is available via PAM modules or via new security daemon SSSD. This work analyses the use of RADIUS protocol for user validation and as a part of this work was developed SSSD module which uses this protocol.

Klíčová slova

SSSD, RADIUS, PAM, NSS, přihlašování, autentizace, bezpečnost

Keywords

SSSD, RADIUS, PAM, NSS, login, authentication, security

Citace

Ondřej Hujňák: Support for RADIUS protocol in SSSD, bakalářská práce, Brno, FIT VUT v Brně, 2013

Support for RADIUS protocol in SSSD

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Zeleného a že jsem uvedl veškeré literární prameny a publikace, ze kterých jsem čerpal.

.....

Ondřej Hujňák

May 15, 2013

Poděkování

Chtěl bych poděkovat Ing. Janu Zelenému za cenné rady ohledně tématu jak po stránce technické, tak i po stránce obsahové a odborné. Dále bych chtěl poděkovat komunitě vývojářů podílejících se na vývoji SSSD za pomoc při vývoji a konstruktivní připomínky, zejména pak panu Ing. Jakubovi Hrozkovi.

© Ondřej Hujňák, 2013.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1	Introduction	3
2	Use case	4
3	Linux user login	5
3.1	History	5
3.2	Name Service Switch	6
3.3	Pluggable Authentication Modules	6
4	RADIUS	8
4.1	Overview	8
4.2	Protocol	8
4.2.1	Code	9
4.2.2	Identifier	10
4.2.3	Authenticator	10
4.2.4	Attributes	10
4.3	Library	11
4.3.1	Libradius	11
4.3.2	FreeRADIUS	11
4.3.3	Conclusion	11
5	SSSD	12
5.1	Overview	12
5.2	Architecture	12
5.2.1	Clients	13
5.2.2	Responders	13
5.2.3	Providers	14
5.3	Configuration	14
6	Design	15
6.1	Environment	15
6.2	Architecture	15
6.2.1	Interfaces	16
6.3	Talloc and Tevent	17
6.3.1	Talloc	17
6.3.2	Tevent	17
7	Implementation	18

7.1	Configuration options	18
7.2	Initialization orientated processes	19
7.3	Request handling processes	20
7.4	Source codes	21
8	Testing	22
8.1	Test case 1	22
8.1.1	Description	22
8.1.2	Test progress	23
8.1.3	Test results	24
8.2	Test case 2	24
8.2.1	Description	24
8.2.2	Test progress	25
8.2.3	Test results	25
8.3	Test case 3	25
8.3.1	Description	25
8.3.2	Test progress	26
8.3.3	Test result	27
8.4	Test case 4	27
8.4.1	Description	27
8.4.2	Test progress	27
8.4.3	Test result	27
9	Evaluation	28
9.1	Future directions	29
10	Conclusion	30
A	Setting up the Environment	32
A.1	Installation of Fedora	32
A.2	Installation of SSSD	32
A.3	Installation of LDAP	33
A.4	Installation of RADIUS server	34
B	Attached disc structure	35

Chapter 1

Introduction

RADIUS (Remote Authentication Dial In User Service) server is often used to manage access to various services such as internet access (PPP) [6], tunnels etc. It grants or denies access based on user credentials stored in data backend which can vary depending on the configuration. SSSD (System Security Services Daemon) is service that authenticates users against different sources called providers which currently cannot use RADIUS server as a provider to authenticate users and the target of this work is to allow SSSD use RADIUS server to do so.

That will allow users to come to a computer with configured SSSD, type in their RADIUS username and password and be granted access to system in the same way as if they have been using system credentials. Having all users stored in one database makes it easier for system administrators to manage users in enterprise solutions, because they only have to take care of a single user database, which is used by RADIUS server and control access to all computers with SSSD and all other services that can make use of RADIUS.

The resulting module will be able to authenticate users on Linux machines with running SSSD. For proper functionality RADIUS server with data backend compatible with SSSD must be present in production environment and data backend have to be able to supply all user account data needed by system directly to SSSD, because resulting module do not provide this functionality.

The use case of this module is described in chapter 2 together with diagram of entities that take part in user login and communication between them. The process of user login in operating system Linux is then outlined in chapter 3. Analysis of RADIUS protocol and SSSD is stated in chapters 4 and 5. Chapters 6, 7 and 8 describes development of SSSD module for RADIUS protocol from the design part to testing of implemented module.

Chapter 2

Use case

Typical use case of this module consists of some user database that stores remote user accounts, RADIUS server and SSSD. SSSD is configured to fetch identity information from user database and authenticate against RADIUS server. RADIUS server is configured to use user database as data backend. RADIUS server might be standalone or a part of some more advanced deployment that uses RADIUS protocol to communicate with clients (e.g. some OTP¹ solutions).

When correctly set up, every authentication of user stored in user database is transparently passed to RADIUS server and response is returned back in the same way as local authentication.

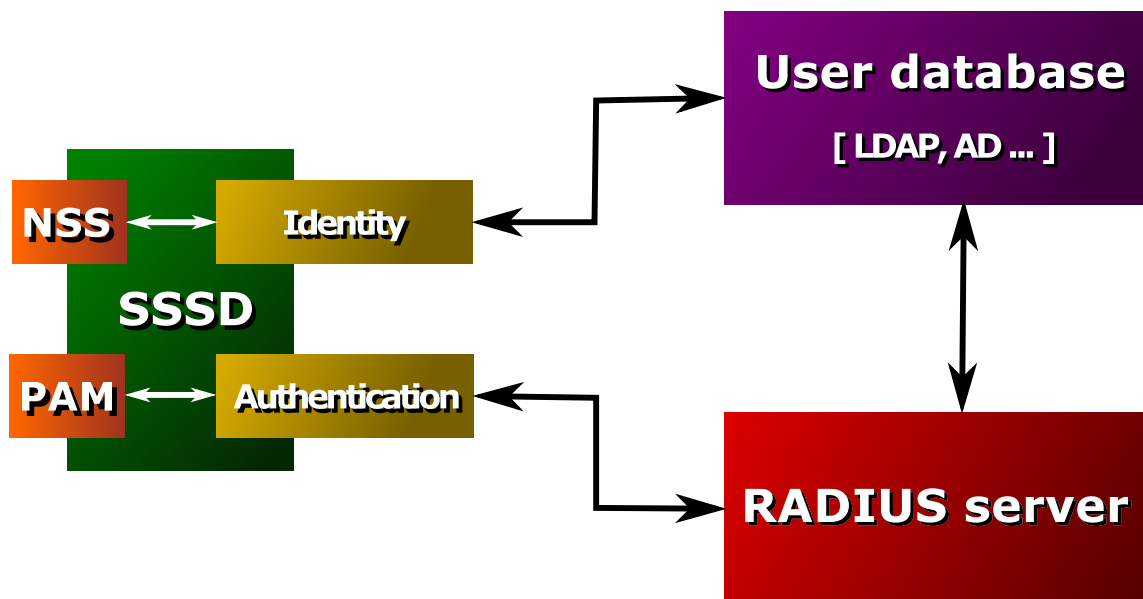


Figure 2.1: Use case architecture

¹One-time Password

Chapter 3

Linux user login

This chapter describes processes during user login in operating system Linux. The aim of this work is to create a module, which will allow remote user login, thus understanding those processes is essential for further advancement.

User login can be divided into four phases – identity, authentication, authorization and session. The task of the first phase is to check, whether a user exists and load all available information about the user. In the second phase user proves (usually by typing password), that he owns the user account loaded in the first phase. When authenticated, system checks if the user has sufficient rights for the requested action (i.e. login to host). If the user has passed all three phases, login is allowed and new session is created for the user.

3.1 History

It is well known for the Unix-like systems, that they store user data in special file */etc/passwd*. This file originally contained everything needed to login user such as login, password hash, identifiers etc. and every program that needed to validate user or fetch user data read it directly from this file [8]. This file had to be public, so that everyone could pair IDs to logins and other user information, but exposed password hashes caused security risks. That is why passwords were moved to different file – */etc/shadow*, which is readable only by root (administrator).

This change with the change of cryptographic function meant rewrite and recompile every single program that used user validation. To abstract access to these information and allow easy changes in password storage was developed PAM (Pluggable Authentication Modules). Not only PAM provides interface to access user passwords and rights, but it is also highly configurable which allows better access management.

With the spread of information technology enterprise productions started to use centralized user storage. To unify access to these storages was created NSS (Name Service Switch), which abstract different sources of information and present them over single interface.

3.2 Name Service Switch

The Name Service Switch (NSS) provides access to different nameservices over unified interface. It can be configured to return data from local files, database, remote resource or even multiple sources combined. NSS was originally developed by Sun Microsystems for their Unix operating system – Solaris, later was developed the same service for Linux [3].

NSS groups information to 11 basic internal databases [3], every database is designed to contain specific information and can be configured with different sources. Information for the identity services is covered by three of them – *passwd* database contains the same information as in */etc/passwd* file (i.e. login, IDs, shell ...), *shadow* database contains user passwords and *group* database contains information about groups (name, ID). Apart from databases designed for login services, NSS provides databases for network services (*services*), host names (*hosts*), networks (*networks*) etc.

As was stated before, every database can have specified multiple sources. Available sources depends on system configuration, but *files* source is always present. This source reads data from local files such as */etc/passwd*, */etc/hosts* etc. Another common sources are *db* which reads information from database, *dns* loads data from Domain Name Service servers and *nis* uses Network Information Service. Source using SSSD to obtain data is called *sss*.

NSS uses */etc/nsswitch.conf* as its configuration file. Configuration settings of one database is written on exactly one line, that consists of database name followed by list of sources in the order in which they should be used. Example of configuration (just a part):

```
passwd: files sss
shadow: files sss
group:  files sss

hosts:  files dns
```

3.3 Pluggable Authentication Modules

Pluggable Authentication Modules (PAM) is service that abstracts login oriented procedures and provides unified interface to access them. PAM is highly configurable and modular, which means that PAM consists of multiple parts that are loaded ad hoc in the runtime. PAM allows to set different configuration schemes for various programs, which grants administrator full control over login processes in the system.

PAM provides four separate tasks – authentication management (*auth*), account management (*account*), session management (*session*) and password management (*password*) [10].

auth Purpose of this module is to authenticate user. It verifies user by some mean of identification (e.g. password) and it can grant some privileges or group memberships to the user.

account This module is designed for authorization, it permits (or restricts) access based on some rules (e.g. system resources, access control list).

session This module processes all things that precedes and follows after the given service such as logging etc.

password This module is used to update authentication token, usually every challenge-response authentication method have password module. This module is also used to change user password.

PAM configuration files are stored in `/etc/pam.d/` directory, every PAM aware program can have its own PAM configuration file stored there. Besides that there are some special configuration files – *system-auth* used for system login and *other*, which is used when no program specific configuration is found. PAM configuration files consists of settings for different modules, where every module is specified on one line. The order in which are modules entered in configuration file is preserved when processing given tasks. Each line keeps the following form [10]:

```
type control module-path module-arguments
```

Where *type* is type of task (auth, account ...), *control* specifies how should PAM handle this module (what should happen if the module fails, succeeds). For more descriptive list of control attributes see [10]. *Module-path* specifies path to the module and *module-arguments* are module specific settings.

In the figure 3.1 is shown PAM architecture during authentication process. Application requires authentication and asks PAM to authenticate user. PAM reads configuration and loads desired modules, then passes authentication to loaded modules in the order they are present in configuration file. The result of authentication is returned to the application. PAM provides separated way to exchange textual information by the use of *conversation* function.

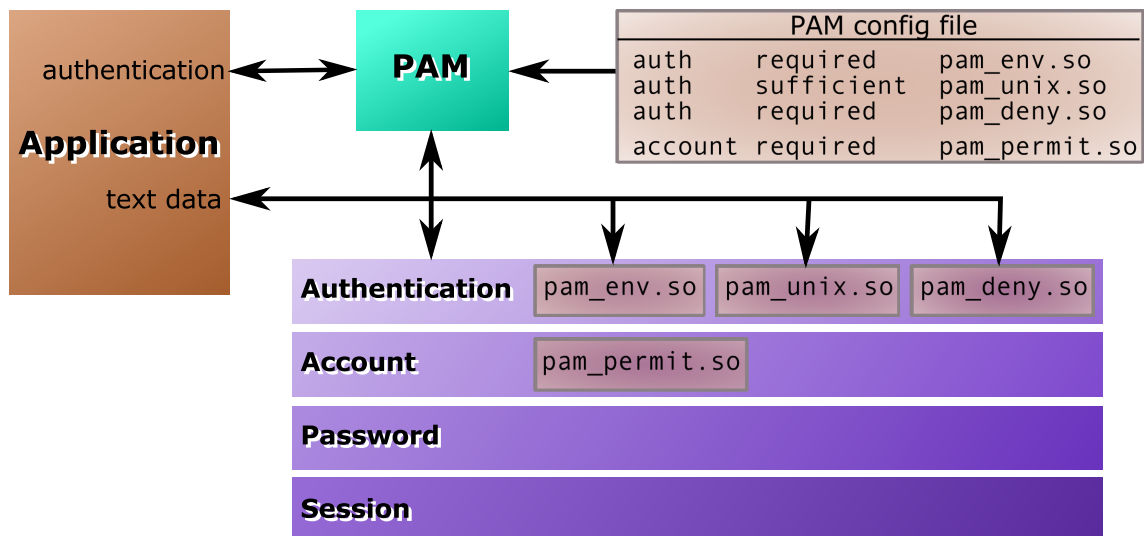


Figure 3.1: PAM architecture

Chapter 4

RADIUS

4.1 Overview

RADIUS (Remote Authentication Dial In User Service) is networking protocol used for AAA (Authentication, Authorization, Accounting) services. Despite having many issues [9] it became de-facto standard for remote authentication and is used in many production environments. RADIUS protocol is an integral part of port security standard IEEE802.1X used by ISP's¹ to control network access.

RADIUS protocol is based on client-server model, which means, that subjects can take one of two roles – *client* requests some service and contacts server to provide it, *server* provides some service to one or many clients. RADIUS protocol does not define any server-initiated messages, which means that every action have to be initiated by client. Server only waits and when there is some client request it processes it and return requested information (e.g. configuration settings).

Client can be directly some host, or it can be a NAS (Network Access Server), which acts as a gateway to some resource. When host wants to access this resource it connects to NAS, NAS validates it against RADIUS server and grants/denies access.

RADIUS can be divided into two independent parts — one covers authentication and authorization and other one accounting. In this document I will focus on the first part described by RFC2865 [12] as this part is used in the resulting module, if you are interested in RADIUS accounting you can find it in RFC2866 [11].

4.2 Protocol

As I have mentioned before RADIUS is client-server based protocol. In the beginning server used UDP² port number 1645 for authorization requests and 1646 for accounting requests. Because those ports were unofficial and caused conflicts with diametrics, IANA³ officially registered UDP 1812 for RADIUS authentication and UDP 1813 for accounting. The use of UDP suggests that RADIUS is connectionless protocol which is right, moreover RADIUS is also sessionless which means, that every request is handled as a new one and no data about connected clients are stored.

¹Internet Service Provider

²User Datagram Protocol

³Internet Assigned Numbers Authority

RADIUS messages are always encapsulated in exactly one UDP packet. You can see structure of that packet in the figure 4.1.

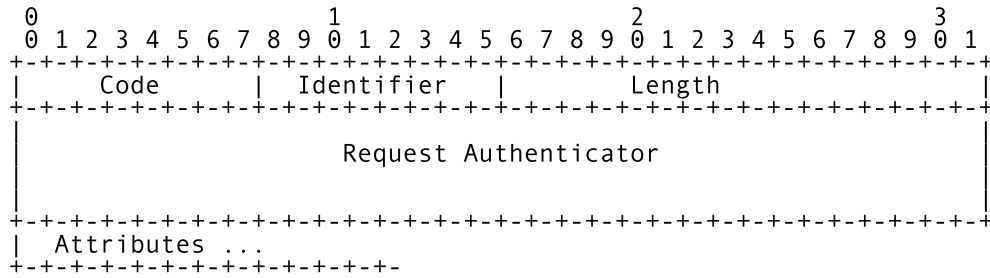


Figure 4.1: RADIUS packet

4.2.1 Code

Code specifies type of the packet. Access part of RADIUS protocol uses these 4 codes:

- 1 Access-Request
- 2 Access-Accept
- 3 Access-Reject
- 11 Access-Challenge

Every RADIUS action begins with client sending Access-Request packet to RADIUS server. Server validates user against data backend and returns Access-Accept in case the validation is successful, or Access-Reject otherwise. Or server can send Access-Challenge packet which requests additional information. In that case client have to generate another Access-Request with the proper answer. You can see the flow diagram in the figure 4.2.

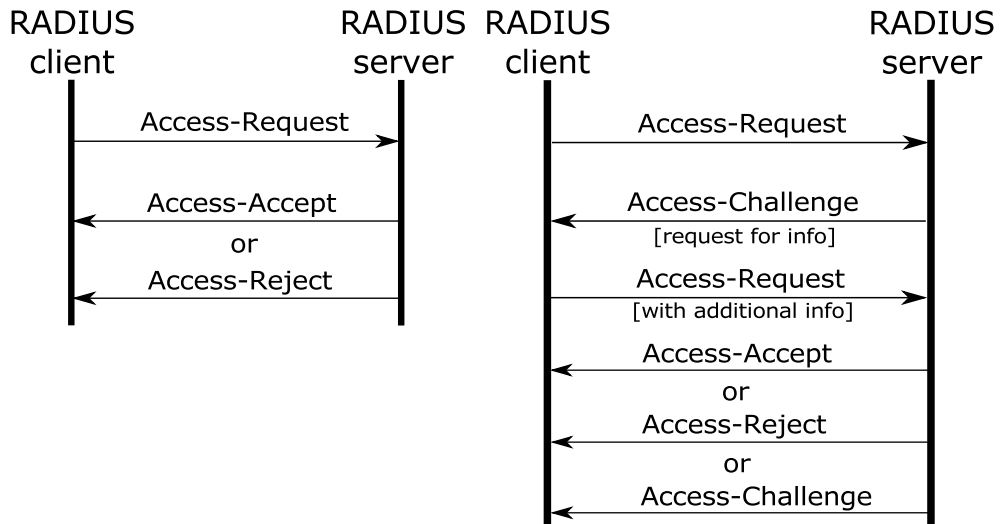


Figure 4.2: RADIUS flow

4.2.2 Identifier

Identifier is a number used for matching requests with replies and detecting duplicates. Client generates identifier number when sending request either randomly, or from some counter and server sends reply with exactly the same identifier.

4.2.3 Authenticator

This field is used to encrypt user password and authenticate reply from RADIUS server. It contains completely different information in request and response packets. In request packets it contains random number which is used to encrypt user password and should be unique to prevent anyone decrypting it. In response packets it contains MD5 hash of the message which ensures integrity of the message.

4.2.4 Attributes

List of attributes that can vary in length and in response packets can be even omitted. I will mention only some attributes related to this work, full specification is available in rfc [12]. Every attribute is represented by type number and have similar structure which you can see in the figure 4.3.

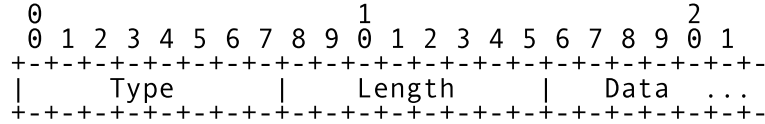


Figure 4.3: General RADIUS attribute

User-Name [1]

In data part is stored a string with user name which is to be validated by server.

User-Password [2]

Data part contains string with encrypted user password. The cipher is counted from shared secret S , RADIUS authenticator RA and user password p by the following algorithm:

$$\begin{aligned}
 b_1 &= MD5(S + RA) & c(1) &= p_1 \text{ xor } b_1 \\
 b_2 &= MD5(S + c(1)) & c(2) &= p_2 \text{ xor } b_2 \\
 &\vdots & &\vdots \\
 b_i &= MD5(S + c(i-1)) & c(i) &= p_i \text{ xor } b_i
 \end{aligned}$$

Where $+$ denotes concatenation and $c(1) \cdots c(i)$ are characters of encrypted string.

Service-Type [6]

Contains number representing a requesting service such as:

- 1 Login** – user tries to connect to host
- 8 Authenticate only** – do not try to authorize user

NAS-Identifier [32]

String in data part identifies NAS that sent Access-Request. When sending Access-Request NAS have to identify itself either by this identifier or by IP address stored in NAS-IP-Address attribute.

4.3 Library

The constant problem of RADIUS protocol is its implementation. Unfortunately it's common that every project implements this protocol from scratch instead of using some library which leads to frequent flaws in different implementations. Even if the protocol is rather simple I decided to use a library to enforce safety of this module. There are two major libraries for RADIUS protocol – libradius and freeRADIUS.

4.3.1 Libradius

This library was developed by Juniper Networks Inc. that holds copyright, but the use is permitted if the copyright is preserved. It is synchronous library for developing RADIUS clients with simple API⁴ and short documentation covered in one man page. However in Fedora this library is packed only as libxradius, which is version adjusted for use with Apache and needs to link with Apache libraries (APR) which is unacceptable.

4.3.2 FreeRADIUS

FreeRADIUS is open source project that covers RADIUS server, server library, client library, PAM and apache modules. Development of freeRADIUS is sponsored by Network RADIUS Inc. which offers its own proprietary libraries and server as well. Client library is available under BSD license so that everyone can use it, it is synchronous library with more complex API than libradius, but the documentation is rather missing. Links on the project page [1] leads to nonexistent pages, so the only source of information are examples packed with library and header file. In Fedora there is only a fork of this library available – radiusclient-ng. This fork is newer than original freeradius-client and unlike the original one it's still maintained, so according to [1] FreeRADIUS project adopts this fork. Unfortunately this fork completely omits all functions to set configuration programmatically and leaves only one function `rc_read_config` that loads configuration from file, which is unpleasant because I need different style of configuration file.

4.3.3 Conclusion

As we can see, both libraries are unsuitable for this module in the packages that are available for Fedora. Libradius requires another libraries which would cause unwanted dependencies and the only way to use FreeRADIUS is to create temporary file with configuration. Another option is to write RADIUS client communication directly without any library or bundle whole library in source codes which would cause problems in maintainence.

⁴Application Programming Interface

Chapter 5

SSSD

5.1 Overview

System Security Services Daemon (SSSD) [2] is a system daemon that provides identity, authentication and authorization of local and remote users via common framework. SSSD allows you to transparently connect to your system with your credentials stored in LDAP¹, Active Directory or different identity resource. Even more, thanks to caching of user data, SSSD is able to provide services even if remote resource is currently unreachable. SSSD is developed since September 2008 by community led by Red Hat Inc. together with FreeIPA² project that covers complementary services to SSSD (mainly identity management).

SSSD connects to system over NSS and PAM modules, every request sent to those modules is forwarded to SSSD, processed internally and the result is sent back over the same interface. Besides that, since version 1.8 SSSD integrates with some third party applications as well. SSSD allows administrator to set multiple identity resources called domains. Every domain consists of three service groups that corresponds with login phases described in chapter 3 – identity oriented services, authentication services and authorization services.

The main advantages of using SSSD instead of various PAM modules are easy and powerfull configuration stored in one file – `/etc/sss/sss.conf`, that allows to configure multiple domains, good server oriented functions and unique use of SysDB database. Thanks to SysDB SSSD allows to cache user data from remote resources and in case of resource unreachability use SysDB to provide services (including authentication). SSSD detects if remote servers are available and reacts to change of its status dynamically.

5.2 Architecture

SSSD architecture is modular and multiprocess. Main process called monitor is responsible for spawning modules that are needed and restarting them, if they are unexpectedly ended. Modules can be divided into three groups — clients, responders and providers. In the figure 5.1 is visible this division and communication between modules. Communication between SSSD modules is ensured by Unix pipes. Communication between monitor and modules is

¹Lightweight Directory Access Protocol

²IPA stands for Identity Policy Audit

encapsulated into the subset of DBus messages, but other communication uses only very simple binary protocol.

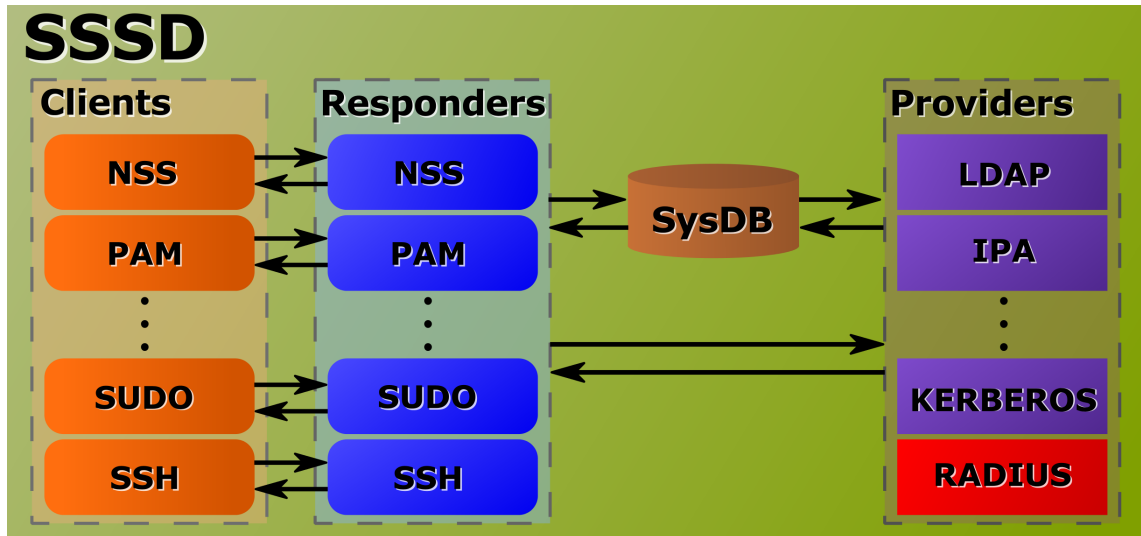


Figure 5.1: SSSD architecture

5.2.1 Clients

Clients create entry point to SSSD as they are modules, which contain interface for communicating with desired application on one side and are connected to SSSD on the other one. Task of every client is to work closely with its application, receive its requests, convert them into SSSD data structures and forward them to their responders. Clients and responders work very closely, which is why every client usually has its own responder, which is designated exclusively for that client. SSSD contains client modules for PAM and NSS and some third party applications like sudo or ssh.

5.2.2 Responders

Responders provide SSSD services to clients. When there is any type of request, it comes to responder which acts upon the request and tries to solve it. Responders can use two types of data sources – SysDB database and providers. Responder gets the data from those data sources, processes it and transforms it to data structures that can be sent back to clients. Every responder works with different data according to client that is paired with it and performs different operations to satisfy the client needs.

PAM responder

Because PAM responder handles all requests from PAM service, which includes authorization of users, this is the responder that will be using RADIUS provider as data backend. PAM responder gets data from PAM client that loads them from internal PAM structure into `pam_items` structure. PAM responder repacks those data to different structure – `pam_data`, which contains a few other items like `pam_status` to detect state of action and

`offline_auth` to detect offline authentication. Because authentication and authorization tasks are highly dependant on the type of auth resource, PAM responder can't solve it on its own and calls provider which gets `pam_data` structure. Response from provider is then sent back to PAM client.

5.2.3 Providers

Finally there are providers that create the backend of SSSD. Providers acts as data resources and provides services for responders. Every provider can provide services from one or more groups mentioned in overview chapter 5.1 and every provider is specific for one type of remote resource (currently are present providers for Active Directory, IPA, Kerberos, LDAP and local one called simple). When provider gets a request from responder it consults it with remote resource over specific protocol and returns results back to resolver repacked in internal structures.

5.3 Configuration

As was mentioned in overview section 5.1 configuration file is stored in `/etc/sss/sss.conf`. This file contains all configuration options for SSSD divided into sections, where main section is called `sss`. Section `sss` have to contain enumeration of services (corresponds with responders) and domains. Every domain has also its own section, where are defined domain specific configurations including providers that should be used for given domain. Every domain have to contain `id_provider`, which provides identity services, but all other providers are optional. Another one is called `auth_provider` and handles authentication tasks. If there is no `auth_provider` set, SSSD will try to use `id_provider`. Last provider I will mention is `access_provider` that is used for authorization and if it is not set it will permit every user.

Sample configuration could look like this:

```
[sss]
domains = my.domain
services = nss, pam
config_file_version = 2

[my.domain]
id_provider = ldap
ldap_uri = ldaps://ldap.my.domain
ldap_search_base = dc=my,dc=domain
```

Chapter 6

Design

6.1 Environment

Production environment for RADIUS provider consists of SSSD capable machine running operating system Linux, RADIUS server and LDAP server. For development purposes was chosen Fedora distribution in current version (18) installed as virtual host, because it supports SSSD in default configuration. Fedora repositories contain FreeRADIUS server, this server was installed and configured to use LDAP server for user database. For LDAP server was chosen OpenLDAP, which is also present in Fedora repositories. For description of installation of production environment see appendix A.

For RADIUS client part was finally chosen krad library [7], which is short of Kerberos RADIUS. It was not mentioned in section 4.3, because it is not a standalone library, but a part of Kerberos project. This library provides needed functionality plus it is asynchronous library, which is big advantage. Drawbacks are that this library is a part of the Kerberos project and therefore depends on Kerberos. Another disadvantage is that this library is brand new and API and ABI¹ are still unstable, but I was assured that ABI compatibility will be preserved. However those drawbacks were discussed and acknowledged by SSSD project leader. This library is written with the use of *verto* library, which is library that abstracts asynchronous event loop and allows users to use different event loop module in the runtime.

6.2 Architecture

Module gets request from *data provider backend*, processes it and sends **Access-Request** packet to RADIUS server. When module gets response from server, it checks the packet code and when it is **Access-Accept** returns **PAM_OK** to back to data provider backend, in other cases it returns **PAM_PERM_DENIED**. Challenge-response authentication is not supported by this module.

¹Application Binary Interface

6.2.1 Interfaces

SSSD-side

SSSD-side interface consists of three functions – `init` function for RADIUS provider initialization, handler function that handles requests and terminating function that returns result of current request to data provider backend.

Init function is first function to be called and its purpose is to initialize all provider structures. This function creates RADIUS provider context that holds all important information for this provider.

Handler is internal module function that have to be registered to data backend during provider initialization. This function takes request packed in `be_req` structure as its parameter. In `be_req` is stored context of whole data provider context and `pam_data` which were sent from responder.

Terminating function is one of the data provider backend functions available for modules. This function finalizes request and sends back to data provider information about errors and authentication status. Those information is later forwarded to the responder.

RADIUS-side

RADIUS-side interface consists of protocol messages described in section 4.2. This interface is covered by the use of *krad* library. This library provides `krad_client_send` function to send `Access-Request` packet and allows to define callback which is called when response is received or if the request timeouts.

`krad_client_send` takes all needed data such as RADIUS server address, attributes, timeout and callback that should be called once reply is received.

Callback gets both request and response packet that was received from the server and allows module to check values in both of them and react accordingly.

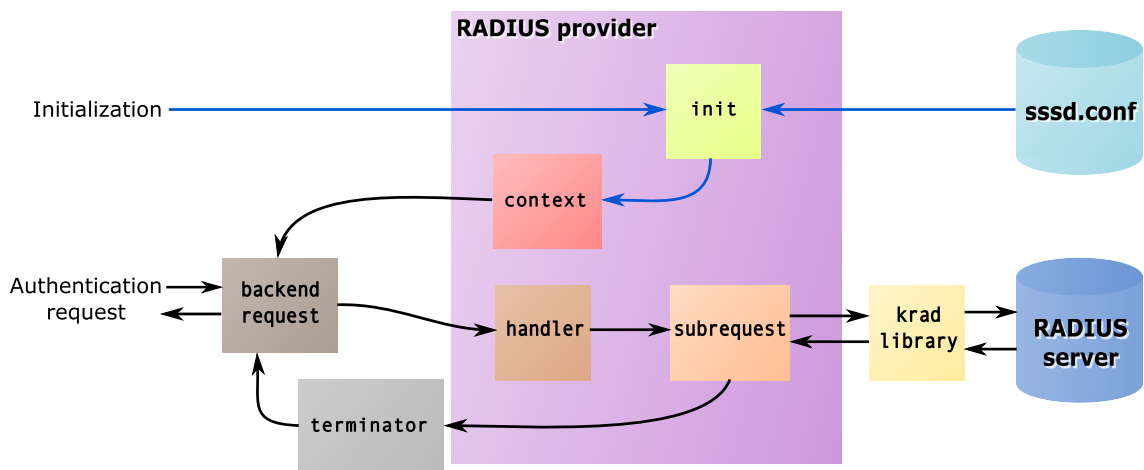


Figure 6.1: Architecture of RADIUS provider module

6.3 Talloc and Tevent

SSSD is designed to be event based program. It means that program consists of event handlers that are associated with particular events. When that event occurs, handler is triggered to react to the situation. In the meantime, when there are no events, program waits in the main loop.

Because SSSD is written mainly in programming language C which doesn't support event based scheme by default, SSSD uses two libraries that originally comes from *Samba* project - *talloc* and *tevent*. Talloc is library for memory management and Tevent is an event system based on Talloc library.

6.3.1 Talloc

Talloc is library for memory management, that uses hierarchical memory pool [4]. Keeping all allocations in a tree structure allows programs to easily free complex structures at once. Another advantage of Talloc is support for destructors.

Talloc uses contexts to keep internal data about allocations and for every allocation talloc must be given context to which could be allocation added (in a tree structure it will be parent node). The main difference between `malloc` and `talloc` is that after allocation `talloc` does not return just a memory pointer, but returns talloc context, that can be used as a parent node in next allocation. When freeing allocated memory, talloc frees all child nodes as well.

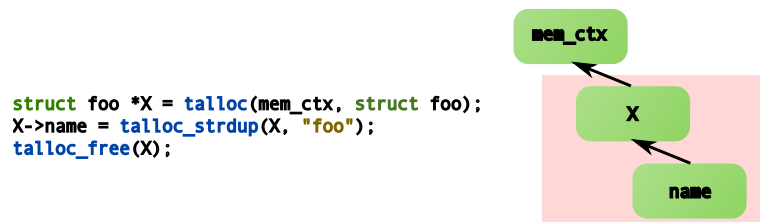


Figure 6.2: Example of talloc usage

6.3.2 Tevent

Tevent is a library for event based programming that uses talloc for memory management [5]. This library provides many event types such as timers and file descriptors and allows to create asynchronous requests.

To create asynchronous request tevent provides `tevent_req` structure, which can be created with function `tevent_req_create()`. Every request can have state structure that holds local variables for whole request and for every request can be set callback that will be called when request is done. Because asynchronous requests are often hierarchical and can make code really confusing, some conventions were stated. Every asynchronous request has a name that describes the aim of the request, then starting function is called `$name$_send`, state structure is called `$name$_state` and callback is called `$name$_done`.

Chapter 7

Implementation

The module was named `rad`, which is short of RADIUS and `rad` prefix is used for module specific objects. Whole module is divided into three source files – `rad_auth.c` contains functions to handle authorization requests, `rad_common.c` is intended for general use and `rad_init.c` ensures provider initialization. Source files are connected by two header files – `rad_auth.h` with authorization specific functions and `rad_common.h` with general purpose functions and one header file with default options – `rad_opts.h`.

7.1 Configuration options

Module is configured in common configure file for whole SSSD - `/etc/sss/sss.conf` and all RADIUS specific options contain prefix `rad`. To use RADIUS for remote authentication is needed to specify `rad` as provider of authentication service by setting for domain:

```
auth_provider = rad
```

This will load `rad` module and every authentication requests will be passed to this module.

Every `rad` provider have to contain some options to determine where is running RADIUS server against which should be users authenticated and shared secret between this server and your host. They are entered in those options:

rad_server contains server which should be used for RADIUS authentication. Can be defined either by IP, by hostname or even by Unix socket.

rad_secret is shared secret between host and server. This have to be the same in the server configuration, or authentication requests will be discarded.

Other options are just optional and can specify some aspects of module behaviour.

rad_port defines port on which is server listening. Can be set by port number, or by name of service (from `/etc/services`). Default value is “radius”.

rad_timeout sets timeout for single response from server in mikroseconds. If server does not reply in time, duplicate request is sent. Default value is 10000.

rad_conn_retries contains number of connection retries. Provider sends up to this number requests and if it doesn't get any reply, authentication is rejected. Default value is 3.

rad_identifier can set an identification that should be sent to server as NAS-Identifier. Default value is determined from hostname.

7.2 Initialization orientated processes

Initialization in SSSD is handled by special init function, where name of the function depends on type of provider and its name. Template of the function is `sssm_${name}_${type}_init`, because this module provides only authentication, there is only authentication init function – `sssm_rad_auth_init`. In the figure 7.1 is shown callgraph of initialization of this module, where green function is entry point to initialization, blue functions are implemented in this module and other functions are external.

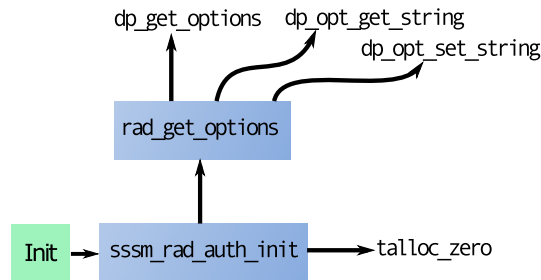


Figure 7.1: Initialization callgraph

Description of functions and structures used for module initialization:

```
int sssm_rad_auth_init(struct be_ctx *bectx, struct bet_ops **ops,
void **pvt_auth_data)
```

This is the main function for initialization, its task is to prepare this module to handle authentication requests. The most important part of this function is creating RADIUS provider context and registering `rad_auth_handler` as a handler for authentication requests. This function gets `be_ctx` structure as a parameter, this structure contains context of whole SSSD domain with all information about it (such as `id_provider`, `auth_provider` etc.). Structure `bet_ops` allows module to register new handler and in `pvt_auth_data` can be stored any data that should be included in every request. In this function is in `pvt_auth_data` stored RADIUS context with important provider-oriented information.

```
struct rad_ctx
```

This structure holds provider context, that contains all provider-specific information. This structure have to be available for every request in this module. Because RADIUS protocol is stateless, it currently holds only provider-specific configuration options.

```
int rad_get_options(TALLOC_CTX *memctx, struct confdb_ctx *cdb,
const char *conf_path, struct dp_option **_opts)
```

This function loads options, checks its validity and return loaded options in `_opts` parameter. `Memctx` is structure allocated with `talloc` that will be used as reference point in other allocations. `Cdb` and `conf_path` are information stored in `be_ctx` that are used to load options.

```
enum rad_opts
```

Enumeration of RADIUS provider options, that is used to access specific option. Last number `RAD_OPTS` is used as a counter of all options.

```
struct rad_options
```

This structure is accessible only within `rad_init.c` file and is used to prevent multiple loading of options. Options are loaded into this structure and next time are reused from this structure instead of redundant option loading.

7.3 Request handling processes

Every request is passed to function set in initialization phase (section 7.2) and ends with the call of the function `be_req_terminate`. In this module, handler function is `rad_auth_handler`. When terminating request, result of operation, pam status and error message (if any) are returned back. In the figure 7.2 si callgraph of an authentication request, where entry and exit points are green, blue functions are implemented by module and other functions are extern.

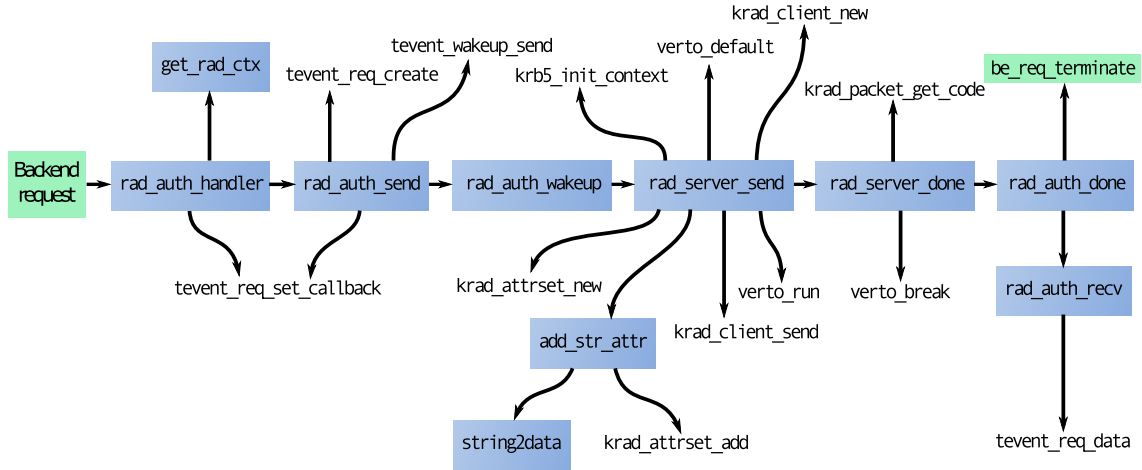


Figure 7.2: Request processing callgraph

Description of functions and structures used for handling authentication requests:

```
void rad_auth_handler(struct be_req *be_req)
```

This is the function that gets called if any request appears and its purpose is to check if it is possible to process this request, create RADIUS request structure `rad_req` and call `rad_auth_send` to create subrequest for inner processing. It takes as a parameter `be_req` structure, which contains data about the request and it sets `rad_auth_done` as a subrequest callback that is called when the subrequest is done.

```
static struct rad_ctx *get_rad_ctx(struct be_req *be_req)
```

This function extracts provider context from request and returns pointer to this context. Because location of provider context in the request depends on the type of PAM command, it also checks if this type of PAM command is supported by rad provider.

```
struct rad_req
```

This structure holds all information needed to process request by rad provider.

```
static int rad_auth_send(struct rad_ctx *ctx, struct pam_data *pd,
    struct be_req *be_req)
```

This function creates subrequest with `rad_state` state structure and sets wakeup function. Wakeup is set to call `rad_auth_wakeup` as soon as possible, which allows program to finish this function and set callback to new subrequest before `rad_auth_wakeup` is called.

```
struct rad_state
```

This is state structure for radius subrequest and contains variables to detect current state of subrequest and all other information needed by subrequest such as kerberos and verto contexts for krad library.

`static void rad_auth_wakeup(struct tevent_req *req)` Only purpose of this function is to call `rad_server_send`. The reason for this solution is need to set a callback before `rad_server_send` is called.

`static int rad_server_send(struct rad_state *state)` This function assembles RADIUS packet with user data and sends it to server for authentication. This function initializes `verto` and `kerberos` contexts, creates RADIUS client with the use of `krad` library and fills it up with user data. Then it passes created client to `krad` library, sets `rad_server_done` to be called once there is an answer from the server and starts `verto` event loop.

`static krb5_error_code add_str_attr(krad_attrset *attrs, const char *attr_name, const char *attr_val)` This helper function adds string `attr_val` to attribute list `attrs`. Parameter `attr_name` defines what RADIUS attribute should be associated with given string.

`static inline krb5_data string2data(const char *str)`
Because `krad` library needs data to be entered packed in Kerberos data structure, this function takes string and packs it so. Because it makes duplicates of every string, it is needed to explicitly free memory after use.

`static void rad_server_done(krb5_error_code retval, const krad_packet *req_pkt, const krad_packet *rsp_pkt, void *data)`
This function gets called if there is response from server or if counter timeouts. It quits `verto`, sets request status according to the response and finishes request. `Retval` variable contain return value from internal `krad` function that receives and processes reply from the server, `req_pkt` and `rsp_pkt` are structures containing request and response packets and in `data` part is stored radius request. If response packet code is `Access-Accept` status is set to `PAM_SUCCESS`, otherwise is set to `PAM_PERM_DENIED`.

`static void rad_auth_done(struct tevent_req *req)` This function finishes the request. It gets called when subrequest is done and finishes the request by calling `be_req_terminate` with the state of subrequest. This state is loaded by calling `rad_auth_rcv` function.

`static int rad_auth_rcv(struct tevent_req *req, int *pam_status, int *dp_err)`
This is just a helper function that loads state of subrequest passed as argument `req` into `pam_status` and `dp_err`.

7.4 Source codes

Whole SSSD project is included on attached disc in the directory `sssd`. Because type of this module is provider, its situated among other providers in `src/providers` subdirectory under its name – `rad`. To compile `sssd` with this module is needed to configure makefile with `--with-radius` or alternatively with all experimental features (`--enable-all-experimental-features`).

Chapter 8

Testing

SSSD allows to specify verbosity of logs for every section of *sssd.conf* and thanks to using masks it is possible to define exactly what should be logged. Allowed value for the mask is in range from 0x0010, which means only fatal failures and is default, to 0xFFFF0 which means to log everything. Masks are created by logical OR of possible values that can be found in manpages for SSSD configuration file (**man 5 sssd.conf**). Example value then could be:

```
[domains/DOMAIN]
debug_level = 0x05f0
```

This mask sets all failures (0x0010 - 0x0080), configuration settings (0x0100) and function trace messages (0x0400). Every section is set separately, so configuration for section [sssd] does not apply for any other sections. This allows more precise configuration of what should be logged and it also corresponds with log files, because every section has its own log file placed in */var/log/sssd*.

8.1 Test case 1

8.1.1 Description

This test verifies, whether user is successfully logged in, when he enters correct user credentials.

Prerequisites LDAP server running with at least one Posix account stored in its database (user test was used in this test). RADIUS server configured to answer requests from testing computer and using LDAP server for user validation. SSSD configured with domain that uses LDAP as id_provider and rad as auth_provider.

Procedure In shell run **su** command to change user to that one defined in LDAP and enter password for LDAP user.

Expected results User should be changed to LDAP user and should be given his default shell.

8.1.2 Test progress

In the figure 8.1 is shown output of terminal, where first command shows, that there is no user *test* in local file */etc/passwd*. Then is user *test* logged in, gets its default shell – *bash* and with command *whoami* is verified, that it's really user *test* who is logged in.

```
--[ 01:30 ]----- phredie -----( ~ )
^andra$ grep test /etc/passwd
--[ 01:30 ]----- phredie -----( ~ )
^andra$ su test
Heslo:
bash-4.2$ whoami
test
bash-4.2$
```

Figure 8.1: Successfull login in shell

In the figure 8.2 is shown network communication during this login. Firstly SSSD tries to connect to IPv6 address of given server, after three unsuccessful attempts it goes to another IP address of the target, which is IPv4. Server is running on this address and returns Access-Accept answer, which means that the user is authenticated.

858	8.453286000	::1	::1	RADIUS	139 Access-Request(1) (id=15, l=75)
859	8.453308000	::1	::1	ICMPv6	187 Destination Unreachable (Port unreachable)
861	8.578569000	::1	::1	RADIUS	139 Access-Request(1) (id=15, l=75), Duplicate Request ID:15
862	8.578584000	::1	::1	ICMPv6	187 Destination Unreachable (Port unreachable)
863	8.703857000	::1	::1	RADIUS	139 Access-Request(1) (id=15, l=75), Duplicate Request ID:15
864	8.703872000	::1	::1	ICMPv6	187 Destination Unreachable (Port unreachable)
865	8.829208000	127.0.0.1	127.0.0.1	RADIUS	119 Access-Request(1) (id=112, l=75)
879	8.838235000	127.0.0.1	127.0.0.1	RADIUS	64 Access-Accept(2) (id=112, l=20)

Frame 865: 119 bytes on wire (952 bits), 119 bytes captured (952 bits) on interface 0	
Linux cooked capture	
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)	
User Datagram Protocol, Src Port: 33124 (33124), Dst Port: radius (1812)	
Radius Protocol	
Code: Access-Request (1)	
Packet identifier: 0x70 (112)	
Length: 75	
Authenticator: 7675895142082e9e5a932ef89feba234	
[The response to this request is in frame 879]	
Attribute Value Pairs	
AVP: l=6 t=User-Name(1): test	
AVP: l=18 t=User-Password(2): Encrypted	
User-Password (encrypted): 0c4a191d10c05e70ea19648d2082777f	
AVP: l=6 t=Service-Type(6): Login(1)	
AVP: l=25 t=NAS-Identifier(32): phredie.ondra.hujnak.cz	

Figure 8.2: Network packet preview from Wireshark

In the preview of packet data is shown User-Name of the user (*test*), its encrypted password, service which we demand is Login and hostname of computer that sent this request - *phredie.ondra.hujnak.cz*. Beside that we can see random generated Authenticator and Packet identifier. In IPv6 requests we can observe, that Packet identifier is constant, which means that those requests are duplicate and if server receives more packets with the same identifier, it handles only first one and dismiss all others.

Now we examine log file for domain RAD that was configured to use rad as an auth_provider. Logs for this domain are stored in */var/log/sss/sssd_RAD.log* and in the figure 8.3 is shown authentication part of log. First there is PAM request to authenticate user *test*, then we can observe that callback is set before *rad_server_send* is called. When

there is an answer, `rad_server_done` breaks verto loop and grants permission for the user test to login (because it received Access-Accept as can be seen in figure 8.2). After result of this operation is sent, request is freed and finished.

```
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [sbus_message_handler] (0x4000): Received SBUS method [pamHandler]
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [be_pam_handler] (0x0100): Got request with the following data
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): command: PAM_AUTHENTICATE
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): domain: RAD
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): user: test
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): service: su
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): tty: pts/3
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): ruser: ondra
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): rhost:
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): authtok type: 1
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): newauthtok type: 0
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): priv: 0
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [pam_print_data] (0x0100): cli_pid: 12123
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [rad_auth_handler] (0x0400): Callback set.
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [rad_auth_wakeup] (0x0400): Calling rad_server_send.
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [rad_server_send] (0x0400): Sending request.
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [rad_server_done] (0x0400): Breaking verto.
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [rad_server_done] (0x0400): Permission granted for user test.
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Backend returned: (0, 0, <NULL>) [Success]
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Sending result [0][RAD]
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Sent result [0][RAD]
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [rad_req_destructor] (0x0400): Destructor freeing req.
(Wed May 8 10:52:17 2013) [sssd[be[RAD]]] [rad_auth_done] (0x0400): Request finished.
```

Figure 8.3: Part of `sssd.RAD.log` file with successful login

8.1.3 Test results

Test was successful, user from LDAP was successfully logged in. Login took longer than local login, because this provider is IPv6 ready and RADIUS server was defined by hostname – localhost. This hostname resolves as IPv6 address `::1` and IPv4 address `127.0.0.1`, where IPv6 has priority. Because RADIUS server is listening only at IPv4 first connection fails and server tries second address which succeeds.

8.2 Test case 2

8.2.1 Description

This test is focused on denial of not authenticated users. It checks, whether user is not logged in with invalid credentials.

Prerequisites LDAP server running with at least one Posix account stored in its database (user test was used in this test). RADIUS server configured to answer requests from testing computer and using LDAP server for user validation. SSSD configured with domain that uses LDAP as `id_provider` and `rad` as `auth_provider`.

Procedure In shell run `su` command to change user to that one defined in LDAP and enter intentionally wrong password.

Expected results User should be denied to change user and error message should be printed. User should remain unchanged.

8.2.2 Test progress

In the figure 8.4 is shown output from the terminal, where is seen that login was really refused.

```
--[ 11:51 ]----- phredie -----( ~ )
ondra $ su test
Password:
su: incorrect password
```

Figure 8.4: Failed login in shell

In network preview in the figure 8.5 are two packets – Access-Request which is sent to the server and Access-Reject which indicates failed authentication request. IPv6 packets sent to the server before IPv4 communication are omitted from the figure as they didn't change from test 1.

```
4422 46.32633800(127.0.0.1 127.0.0.1 RADIUS 119 Access-Request(1) (id=143, l=75)
4591 47.32758300(127.0.0.1 127.0.0.1 RADIUS 64 Access-Reject(3) (id=143, l=20)

Attribute Value Pairs
+ AVP: l=6 t=User-Name(1): test
+ AVP: l=18 t=User-Password(2): Encrypted
+ AVP: l=6 t=Service-Type(6): Login(1)
+ AVP: l=25 t=NAS-Identifier(32): phredie.ondra.hujnak.cz
```

Figure 8.5: Network packet preview from Wireshark

The log file now says “Permission denied” instead of “Permission granted” and returned values are 0, which means that there was no error and 6, which means that user was not authenticated.

```
(Wed May 8 11:39:23 2013) [sssd[be[RAD]]] [rad_auth_handler] (0x0400): Callback set.
(Wed May 8 11:39:23 2013) [sssd[be[RAD]]] [rad_auth_wakeup] (0x0400): Calling rad_server_send.
(Wed May 8 11:39:23 2013) [sssd[be[RAD]]] [rad_server_send] (0x0400): Sending request.
(Wed May 8 11:39:27 2013) [sssd[be[RAD]]] [rad_server_done] (0x0400): Breaking verto.
(Wed May 8 11:39:27 2013) [sssd[be[RAD]]] [rad_server_done] (0x0400): Permission denied for user test.
(Wed May 8 11:39:27 2013) [sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Backend returned: (0, 6, <NULL>) [Success]
(Wed May 8 11:39:27 2013) [sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Sending result [6][RAD]
(Wed May 8 11:39:27 2013) [sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Sent result [6][RAD]
(Wed May 8 11:39:27 2013) [sssd[be[RAD]]] [rad_req_destructor] (0x0400): Destructor freeing req.
(Wed May 8 11:39:27 2013) [sssd[be[RAD]]] [rad_auth_done] (0x0400): Request finished.
```

Figure 8.6: Part of sssd.RAD.log file with permission denied

8.2.3 Test results

User was denied to change to LDAP user and “incorrect password” message was displayed. This covers expected results by 100%. Unsuccessful login took a long time from the same reasons as in test 1.

8.3 Test case 3

8.3.1 Description

In this test proves correct behaviour of rad provider in case that RADIUS server does not reply.

Prerequisites LDAP server running with at least one Posix account stored in its database (user test was used in this test). SSSD configured with domain that uses LDAP as id_provider and rad as auth_provider.

Procedure Make sure, that RADIUS server is not running or doesn't answer requests from testing host. Then run in shell command `su` to change user to that one defined in LDAP and enter password.

Expected results User should be denied to change user and error message should be printed. User should remain unchanged.

8.3.2 Test progress

In the output from shell in the figure 8.7 is seen, that user was not logged in and error message says "incorrect password".

```
--[ 13:08 ]----- phredie -----( ~ )
ondra $ su test
Password:
su: incorrect password
```

Figure 8.7: Timeout login in shell

Network analyser whose output is captured in figure 8.8 shows 3 packets to IPv6 server address and 3 packets to IPv4, no response from RADIUS server is returned, only ICMP messages.

1123	4.965036000	::1	::1	RADIUS	139 Access-Request(1) (id=65, l=75)
1124	4.965056000	::1	::1	ICMPv6	187 Destination Unreachable (Port unreachable)
1747	6.216512000	::1	::1	RADIUS	139 Access-Request(1) (id=65, l=75), Duplicate Request ID:65
1748	6.216532000	::1	::1	ICMPv6	187 Destination Unreachable (Port unreachable)
1967	7.467880000	::1	::1	RADIUS	139 Access-Request(1) (id=65, l=75), Duplicate Request ID:65
1968	7.467901000	::1	::1	ICMPv6	187 Destination Unreachable (Port unreachable)
2670	8.719347000	127.0.0.1	127.0.0.1	RADIUS	119 Access-Request(1) (id=245, l=75)
2671	8.719371000	127.0.0.1	127.0.0.1	ICMP	147 Destination unreachable (Port unreachable)
2996	9.970785000	127.0.0.1	127.0.0.1	RADIUS	119 Access-Request(1) (id=245, l=75), Duplicate Request ID:245
2997	9.970812000	127.0.0.1	127.0.0.1	ICMP	147 Destination unreachable (Port unreachable)
3311	11.222225000	127.0.0.1	127.0.0.1	RADIUS	119 Access-Request(1) (id=245, l=75), Duplicate Request ID:245
3312	11.222249000	127.0.0.1	127.0.0.1	ICMP	147 Destination unreachable (Port unreachable)

Figure 8.8: Network packet preview from Wireshark

Finally in the log (figure 8.9) is stated "Request timeout" and returned error state. Number 2 means that request was not completed, because of a timeout and number 4 tells PAM that there was system error during authentication.

```
[sssd[be[RAD]]] [rad_auth_handler] (0x0400): Callback set.
[sssd[be[RAD]]] [rad_auth_wakeup] (0x0400): Calling rad_server_send.
[sssd[be[RAD]]] [rad_server_send] (0x0400): Sending request.
[sssd[be[RAD]]] [rad_server_done] (0x0400): Breaking verto.
[sssd[be[RAD]]] [rad_server_done] (0x0040): Request timeout. No response from server.
[sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Backend returned: (2, 4, <NULL>) [Request timed out (System error)]
[sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Sending result [4][RAD]
[sssd[be[RAD]]] [be_pam_handler_callback] (0x0100): Sent result [4][RAD]
[sssd[be[RAD]]] [rad_req_destructor] (0x0400): Destructor freeing req.
[sssd[be[RAD]]] [rad_auth_done] (0x0400): Request finished.
```

Figure 8.9: Part of sssd_RAD.log file with server timeout

8.3.3 Test result

User was indeed refused access, but before error message was printed user had to wait for a fairly long time. This is caused by waiting for response from server that would never come. This can be adjusted by setting shorter timeout, in the future there should be some state of server present in rad provider, so that user did not have to wait.

8.4 Test case 4

8.4.1 Description

This test case proves that no unknown users can be logged in to the system.

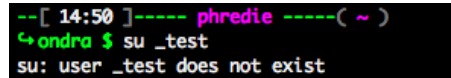
Prerequisites Running LDAP server. SSSD configured with domain that uses LDAP as id_provider and rad as auth_provider.

Procedure In shell run `su` command to change user to non-existent one.

Expected results User should be denied to change user and error message should be printed. User should remain unchanged.

8.4.2 Test progress

In the figure 8.10 is captured output from shell. Error message was printed even before prompt for password, because error was encountered in the first phase of login – identification.



```
--[ 14:50 ]----- phredie -----( ~ )
↳ ondra $ su _test
su: user _test does not exist
```

Figure 8.10: Login of non-existent user

Because login failed before authentication phase, rad provider is not called at all and there are no packets sent at all.

8.4.3 Test result

This test successfully showed that unknown users cannot login to the system.

Chapter 9

Evaluation

The implementation of RADIUS provider uses pretty straightforward architecture and division described in chapters 6 and 7, which allows easy orientation in code for everyone who knows *tevent* library. The structures and functions are designed for good readability and extendability to make it easier to maintain.

For the RADIUS protocol implementation was chosen *krad* library, which has an advantage, because unlike other libraries mentioned in section 4.3 it is asynchronous library. Synchronous actions in SSSD are solved by moving them to different process and using pipes to communicate between asynchronous and synchronous processes, but use of asynchronous library allows to include this library directly into main process and use it as a part of SSSD, which means lower system requirements. This library have many drawbacks stated in section 6.1 caused mainly because this library is brand new and is a part of *kerberos* project, but still it was chosen as the best option.

During implementation was discovered a problem in *verto* library which is used in *krad*, that after use of this library context could not be freed, because *free* function would destroy main event loop and cause provider stop responding to any actions. This problem was discussed with *verto* developers and should be fixed in the next version of the library. Until this is done, *verto* context cannot be freed and causes minor memory leaks.

Currently it is possible to authenticate against RADIUS via PAM modules, one is a part of the FreeRADIUS project [1]. This module provides basically the same functionality as SSSD with this provider and it uses RADIUS accounting to register user login at server-side. The advantage of using SSSD is in better configuration, because everything is stored in one readable file, and great extendibility. Unlike FreeRADIUS, SSSD allows to set specific configuration for every RADIUS server thanks to diversion into domains and can easily combine different account data resources.

RADIUS provider is really minimalistic in comparison with other providers in SSSD. Unlike other providers it provides only one task – authentication and it doesn't have possibility to store user data in cache for offline use. On the other hand this provider is brand new and makes use of current features of SSSD such as specific debug levels, that allows users to define exactly what messages to log and internal error codes that make it easier for programmers to locate the origin of an error.

9.1 Future directions

In the tests was mentioned, that login procedure can in some cases take a long time. To prevent this it would be much more convenient to periodically check server availability and store its status in provider context. When the server is unreachable, provider could act immediately and wouldn't have to wait till the request timeouts.

It would be also pleasant if it was possible to store user credentials with server response in SSSD cache, like it does in some other providers. This would allow provider to check cache instead of remote server once there is an entry, which would reduce network traffic and allow users to login with remote credentials even if server is not reachable.

The addition most wanted by upstream is possibility to map users between RADIUS users and local ones. Having some map would allow users to connect to the computer with RADIUS account that does not have to be in any other resource for identity. For example there will be map, that every RADIUS user is to be connected as *guest* account. User than logs in with RADIUS credentials, but in the system is logged as user *guest* with all permissions and restrictions defined for user *guest* and system does not need to know about RADIUS user.

Another possibility is adding RADIUS accounting to the provider if it is desirable. This enables to log user logins on server side, which could be used for example for monitoring usage of different computers.

Chapter 10

Conclusion

This work analysed the use of RADIUS protocol for remote user login in Unix-like systems. Only one part of RADIUS was proved to be useful and is used only for user authentication. As a result of this work was developed module for SSSD that allows users from some remote resource to be authenticated against RADIUS server.

The main feature of SSSD for a programmer is definitely an event-based concept of whole program. That is, because asynchronous programming is not typical for programming language C in which is SSSD mainly written. However this schema is very convenient for this service, because it corresponds with its use, when daemon waits till it gets request for some task (either from NSS, PAM or different client), solves it and after task is solved it returns answer and waits for another requests. To use this feature in C, which is known as a lower level language (in comparison with object oriented and other very high level languages) SSSD uses tevent library, which is described in section 6.3.

The biggest problem showed up to be the lack of documentation for many open source projects. For example SSSD project have only user-targeted documentation that describes how to install and configure SSSD, but programmer-oriented documentation about internal interfaces and essential objects for writing modules is completely missing. Problem can cause outdated documentation as well, for example LDAP recently moved from *slapd.conf* configuration file to directory of *ldif* files but documentation and many manuals are written for configuration file. Big difference is that in new config style, *ldif* files shouldn't be manually edited, but should be changed in the same way as LDAP internal database (programs like *ldapmodify*). Use of bleeding edge software causes problems not only with the lack of information about it, but can also change its API or contain some bugs.

Unpleasant surprise was a really unsatisfactory situation in RADIUS client libraries. Although there are some, they are not present in target system in an usable way and it seems that they are declining. Thanks to community of SSSD developers I have found brand new RADIUS library, which was developed for Kerberos OTP and used it, but because it's brand new and uses *verto* library, which is pretty new as well, it caused some problems mentioned in chapter 9.

Despite the obstacles the module was successfully developed and is able to authenticate users as requested. Moreover this module is written with regard to current principles in SSSD and is readable and easily extendible. The module was named *rad*, which is short of RADIUS and its source code resides in *src/providers* directory in its own subdirectory *rad*. The module is loaded dynamically by SSSD monitor when it is requested in configuration, description of configuration can be found in section 7.1.

Bibliography

- [1] The freeradius project. <http://freeradius.org/>, [quoted 2013-04-28].
- [2] Sssd - system security services daemon. <https://fedorahosted.org/sss/>, [quoted 2013-04-30].
- [3] System databases and name service switch. https://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html, [quoted 2013-05-02].
- [4] talloc documentation. <http://talloc.samba.org/talloc/doc/html/index.html>, [quoted 2013-05-05].
- [5] tevent. <https://tevent.samba.org/>, [quoted 2013-05-05].
- [6] How Does RADIUS Work? http://www.cisco.com/en/US/tech/tk59/technologies_tech_note09186a00800945cc.shtml, [quoted 2013-05-13].
- [7] npmccallum/krb5. <https://github.com/npmccallum/krb5/commits/otp-master/>, [quoted 2013-05-13].
- [8] Peter Hernberg. User authentication howto. <http://www.faqs.org/docs/Linux-HOWTO/User-Authentication-HOWTO.html>, 2000-02-05 [quoted 2013-05-02].
- [9] Joshua Hill. An analysis of the radius authentication protocol. <http://www.untruth.org/~josh/security/radius/radius-auth.html>, 2001 [quoted 2013-04-24].
- [10] Andrew G. Morgan and Thorsten Kukuk. The linux-pam system administrators' guide. http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_SAG.html, 2010-08-31 [quoted 2013-05-02].
- [11] C. Rigney and Livingston. Radius accounting. <https://tools.ietf.org/html/rfc2866>, June 2000 [quoted 2013-05-04].
- [12] C. Rigney, S. Willens, Livingston, A. Rubens, Merit, W. Simpson, and Daydreamer. Remote authentication dial in user service (radius). <https://tools.ietf.org/html/rfc2865>, June 2000 [quoted 2013-04-24].

Appendix A

Setting up the Environment

A.1 Installation of Fedora

There are multiple ways to install Fedora, but the most common is to use compact disc and boot from that disc. You can download image of Fedora 18 disc from <http://fedoraproject.org/cs/get-fedora> and then burn the image onto disc or mount it into virtual CD reader in our case. Now boot from that CD and when context menu of boot-loader GRUB comes up choose Install. After a while graphical installer Anaconda should show up, follow instructions on the screen and after reboot you should get to fresh installation of Fedora 18.

A.2 Installation of SSSD

Firstly install SSSD package from repository by typing (as a root) `yum install sssd`, that will install sssd and all the dependencies. Now we need to ensure, that system passes pam and nss requests to SSSD. In the config file `/etc/pam.d/system-auth` there should be lines containing `pam_sss.so`, if they are missing, add to correct sections following entries (always on the last but one line):

```
auth          sufficient    pam_sss.so  use_first_pass
account       [default=bad success=ok user_unknown=ignore] pam_sss.so
password      sufficient    pam_sss.so use_authtok
session       sufficient    pam_sss.so
```

This will ensure, that pam requests are processed by SSSD. Now check nss configuration in file `/etc/nsswitch.conf`, if it doesn't contain sss add it as follows:

```
passwd:      files sss
shadow:      files sss
group:       files sss
```

Now add a simple configuration to `/etc/sss/sss.conf` if it is not present. Here you can see example of the simplest configuration:

```
[sssd]
domains = local
services = nss, pam
config_file_version = 2

[domain/local]
id_provider=local
```

And set correct rights to the file by `chmod 0600 /etc/sss/sss.conf`. You should be able to start sssd by typing `systemctl start sssd` by now. If there is no error, SSSD is running correctly, but this setting is rather useless, because it uses the same user resource as standard unix files that are used with higher priority. You can find example of my SSSD configuration that uses new RADIUS provider on the attached disc in config directory.

This gives us SSSD up and running, but for development there are another packages needed. That can be installed by typing:

```
yum install openldap-devel gettext libtool pcre-devel c-ares-devel
dbus-devel libxslt docbook-style-xsl krb5-devel nspr-devel libxml2
pam-devel nss-devel libtevent python-devel libtevent-devel libtdb
libtdb-devel libtalloc libtalloc-devel libldb libldb-devel popt-devel
c-ares-devel check-devel doxygen libselenium-devel libsemanage-devel
bind-utils libnl3-devel gettext-devel glib2-devel
```

```
yum install libcollection-devel libdhash-devel libini.config-devel
libpath_utils-devel libref_array-devel
```

Radius provider depends on kerberos libraries and verto library, which needs at least one verto-module installed. To install verto type:

```
yum install libverto libverto-tevent
```

Kerberos libraries that are currently present in Fedora repositories are obsolete and use a different API, that is why kerberos libraries needs to be installed from koji:

<http://koji.fedoraproject.org/koji/buildinfo?buildID=410384>

A.3 Installation of LDAP

OpenLDAP server can be easily obtained from repository by typing `yum install openldap openldap-servers`. OpenLDAP recently switched its configuration options from file `/etc/openldap/slapd.conf` to directory of ldif files `/etc/openldap/slapd.d/`. Those files are not meant to be modified directly, but have to be configured on the fly, that's why first step is to start OpenLDAP server by typing `systemctl start slapd`. When the server is running its time to make some configuration. Because in standard configuration there is only base schema loaded, first thing we have to do is to add a schema with support of POSIX accounts:

```
ldapadd -Y EXTERNAL -H ldapi:/// -f /etc/openldap/schema/nis.ldif
```

Configuration of slapd now consists of adding ldif files to LDAP database. Two example files with configuration are present on the attached disc in config directory. File db.ldif contains basic setting with new suffix, new root DN and new root password. File base.ldif then creates root node and two sub nodes — one called People for user accounts and one for groups. Add those files by typing following commands, second one will prompt for root password that was set in db.ldif.

```
ldapadd -Y EXTERNAL -H ldapi:/// -f db.ldif
ldapadd -xWD "cn=admin,dc=ondra,dc=hujnak,dc=cz" -f base.ldif
```

After that LDAP is ready to load users and groups. Easiest way to transfer users and groups is to use `migrationtools` to convert `/etc/passwd` and `/etc/groups` contents to ldif files and add them to LDAP database. Another option is to write ldif files manually which allows better control over users and groups that are stored in LDAP. It is also possible to change ldif files generated by `migrationtools` to contain only specific accounts and groups.

A.4 Installation of RADIUS server

Finally we can install FreeRADIUS server with LDAP as data backend. In Fedora this is divided into two packages that can be installed by:

```
yum install freeradius freeradius-ldap
```

Configuration files of RADIUS server are located in directory `/etc/raddb/` and setting of LDAP module is in subdirectory `modules/ldap`. The configuration to set up module to communicate with LDAP directory server running on local machine looks similar to this:

```
ldap {
    server = "127.0.0.1"
    identity = "cn=admin,dc=ondra,dc=hujnak,dc=cz"
    password = testing
    basedn = "ou=People,dc=ondra,dc=hujnak,dc=cz"
    filter = "uid=%u"
    ...
}
```

LDAP module have to be enabled by uncommenting ldap entry in `/etc/raddb/sites-enabled/default`. After this we should be able to start RADIUS server with `systemctl start radiusd`.

Appendix B

Attached disc structure

The disc attached to this thesis contains source codes of rad provider as well as this thesis.

- **config/** – contains examples of configuration files
 - **base.ldif** – creates root node and two subnodes - People and Groups
 - **db.ldif** – contains basic setting of LDAP with new suffix and root DN
 - **sssd.conf** – example of configuration file for SSSD
- **scripts/** – contains helper scripts for environment preparation
 - **add_sssd_to_nss.sh** – adds sssd to NSS configuration file
 - **add_sssd_to_pam.sh** – adds sssd to PAM configuration file
 - **install.sh** – this is wrapper that runs all other scripts in order to prepare whole environment
 - **install_packages.sh** – install needed packages from repository
 - **set_ldap_server.sh** – installs OpenLDAP and modifies its setting to contain POSIX accounts and adds user test with password test
 - **set_radius_server.sh** – installs FreeRADIUS and modifies its settings to verify users against LDAP
- **sssd/** – contains complete SSSD project repository, the rad provider is situated in `src/providers/rad` subdirectory
- **tex/** – contains source files for this report
 - **Changelog** – changelog of template for thesis from FIT VUTBR
 - **Makefile** – makefile for this thesis
 - **appendix.tex** – contains all appendixes of this thesis
 - **body.text** – contains main body of this thesis
 - **cover.tex** – is used only to generate cover of this thesis
 - **czechiso.bst** – czech bibliography style
 - **fig/** – this folder contains all figures used in this thesis
 - **fitthesis.cls** – document style from FIT VUTBR
 - **literature.bib** – list of all sources used for this thesis
 - **thesis.tex** – the main file for compiling, it contains definitions of packages and structure of document
- **thesis.pdf** – this report