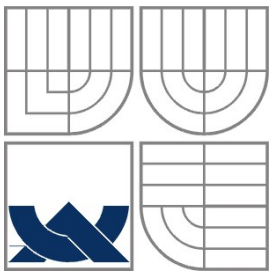
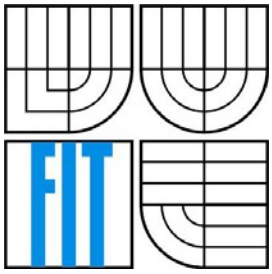


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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 — Ondřej Hujňák
AUTHOR

VEDOUCÍ PRÁCE — Ing. Jan Zelený
SUPERVISOR

BRNO 2013

Zadání:

1. Acquaint yourself with the architecture of SSSD with focus on krb5 back end.
2. Acquaint yourself with the Radius authentication protocol.
3. Design a module for SSSD which will enable using Radius server for authentication.
4. Implement and test the designed module.
5. Evaluate your implementation, its advantages and shortcomings in comparison with existing solutions.

Abstrakt

Tato bakalářská práce se zabývá projektem SSSD, který slouží k identity managementu, autorizaci a autentizaci v operačním systému Linux. Dále se práce zabývá AAA protokolem RADIUS, který poskytuje služby využitelné v projektu SSSD a možností jejich propojení. Cílem této práce je vytvořit modul pro SSSD který umožní ověřování uživatelů vůči RADIUS serveru.

Abstract

This bachelor thesis is focused on SSSD project that provides identity management, authorization and authentication in Linux operating system. This work analyses AAA protocol RADIUS, which provides services that can be used by SSSD and possibility of it's connection. The aim of this work is to develop SSSD module for user authentication against RADIUS server.

Klíčová slova

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

Keywords

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

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

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.

Poděkování

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

Table of Contents

Introduction

RADIUS server is often used to manage access to various services such as internet access, tunnels (PPP) etc. It grants or denies access based on user credentials stored in data backend which can vary depending on configuration. SSSD is service that authenticates users against different sources called providers. SSSD 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. That makes it easier for system administrators to manage users in enterprise solutions, because they only have to take care of a single user database that is used by RADIUS server and control access to all computers with SSSD set and all other services that can make use of RADIUS.

System requirements for the resulting program includes SSSD correctly set up and running on Linux host and RADIUS server with data backend compatible to work with SSSD as ID provider. Development environment consisted of Fedora 18 operating system with FreeRADIUS server and OpenLDAP as data backend for RADIUS and ID provider for SSSD.

Main text

RADIUS

Overview

RADIUS (Remote Authentication Dial In User Service) is networking protocol used for AAA (Authentication, Authorization, Accounting) services. RADIUS uses client/server model, where client is called Network Access Server (NAS). NAS sends requests to RADIUS server and act on the response. Server receives request, authenticates the user and return requested information (usually configuration settings). RADIUS protocol does not define any server-initiated messages.

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, if you are interested in RADIUS accounting it is described in RFC2866.

Protocol

As I have mentioned before RADIUS is client/server based protocol. The server listens on UDP port number 1812 (previously UDP 1645), which means it is connectionless protocol. RADIUS can use either simple authentication (PAP) or use challenge/response authentication (CHAP).

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

in the Illustration 1.

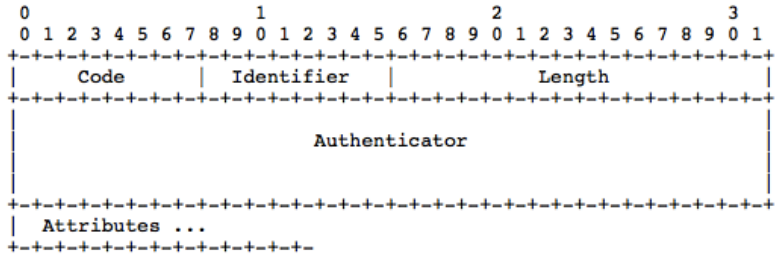


Illustration 1: RADIUS packet

Code

Access part of RADIUS protocol uses 4 codes:

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

Every RADIUS action begins with NAS 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 NAS generates another Access-Request with proper answer. You can see flow diagram in the Illustration 2.

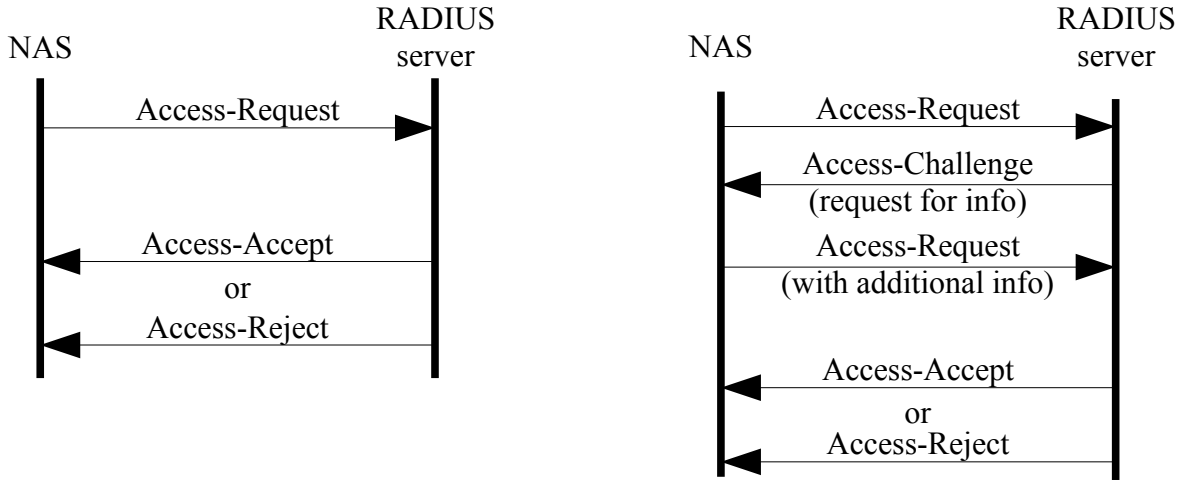


Illustration 2: RADIUS flow

Identifier

Identifier is a number used to matching requests with replies and detecting duplicates.

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 decrypt it. In response packets it contains MD5 hash of the message which ensures integrity of the message.

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. Every attribute is represented by type number and have similar structure which you can see in Illustration 3.

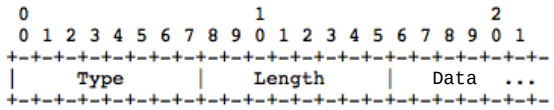


Illustration 3: Generic attribute

User-Name [1]

In data part is stored string with user name which is to be validated by RADIUS 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:

```
b1 = MD5(S + RA)       c(1) = p1 xor b1
b2 = MD5(S + c(1))     c(2) = p2 xor b2
       .                      .
       .                      .
       .                      .
bi = MD5(S + c(i-1))   c(i) = pi xor bi
```

Where + denotes concatenation and c(1)…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.

Library

The constant problem of RADIUS protocol is it's 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.

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 API 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 which is unacceptable.

FreeRADIUS

FreeRADIUS is open source project that covers RADIUS server, server library, client library, PAM library and apache module. Development of freeRADIUS is sponsored by Network RADIUS Inc. which offers it's

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 [3] 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 that original freeradius-client and unlike the original one it's still maintained, so according to [3] FreeRADIUS project adopts this fork. 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.

As we can see, both libraries are unsuitable for this program 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 with program.

SSSD

Overview

System Security Services Daemon (SSSD) is a system daemon that provides authentication, authorization and identity management of remote users via common framework. SSSD allows you to transparently connect to your system with your credentials stored in LDAP, Active Directory or different identity resource. Even more, thanks to caching user data, SSSD is able to provide services even if remote resource is unreachable.

Architecture

SSSD architecture is modular, 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.

Clients requires SSSD services and communicates closely with responders. Usually every client has it's own responder that is designated exclusively for itself. Currently there are three clients included in SSSD project – autofs, sudo and ssh.

Responders provides SSSD services to clients on one side and calls providers to process given task or consults it with internal database. In fact responders creates interface that can be used by clients. Besides responders for internal clients there are pam and nss responders that can be used by any program designed for use with those services.

Finally there are providers that create the backend of SSSD. Their task is to search information in remote or local resources and ensure authentication and authorization of users. Results are being sent to responders and can be stored in internal database for later use. Currently there are providers for Active Directory, IPA, Kerberos, LDAP and local one called simple. Target of this work is to develop a new provider for RADIUS protocol.

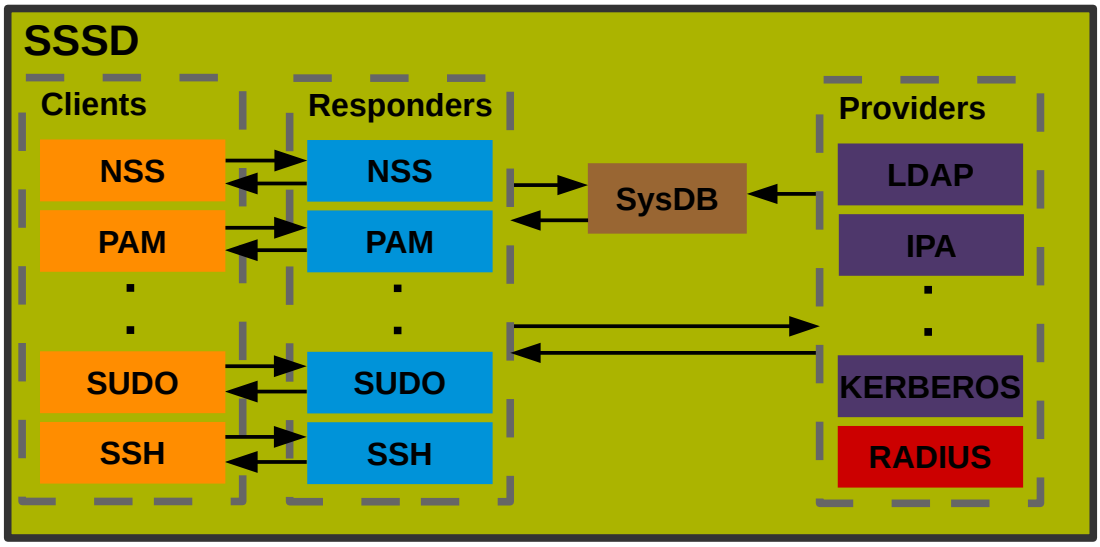


Illustration 4: SSSD architecture

Providers

In this chapter I will focus on providers as the task of this work is to develop one. Providers are data resources that provides services that can be divided into three groups – identity management, authentication and authorization. Not all providers have to cover all these groups, if provider doesn't provide some services, another provider can be set to cover them or default SSSD actions apply. In SSSD configuration file located in /etc/sssd/sssd.conf there are domains that represents resources and every domain consist of at least one provider. Every domain have to include id_provider that covers identity management and it's main purpose is to supply information about user such as username, real name, default shell etc. Next group is authentication and this group is covered by auth_provider. If there is no auth_provider set authorization requests are sent to id_provider. Last group is authorization and in configuration it is in access_provider, if there is no access_provider set every authenticated user is automatically granted permission. Every domain can consist of up to three different providers, where every provider covers one group of services.

Preparing the environment

As was stated in introduction, operating system used for development was Fedora 18 installed as virtual host. Fedora supports SSSD and contains it in default desktop installation, but in custom installations or different Fedora presets (called Spins) it is possible that SSSD will have to be installed and configured. Next thing needed for this work is RADIUS server against which will be users authenticated. As Fedora repositories contains only FreeRADIUS this is the one used. Because RADIUS protocol doesn't provide all information about user needed by SSSD such as default shell and home directory, another resource is needed to provide this information. LDAP is example of suitable resource and it's advantage is, that it can be used by RADIUS server as data backend and SSSD as id provider at the same time. That creates only one user database which is easy to maintain. OpenLDAP is implementation of LDAP that is packed in Fedora repositories.

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 bootloader 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.

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 not add to correct sections 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/sssd/sssd.conf if it is not present. Simplest configuration looks as follows:

```
[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/sssd/sssd.conf`. Now you should be able to start sssd by typing `systemctl start sssd`. 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 in Annex A.

For SSSD development another packages are needed, they 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 libselinux-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 present in Fedora repositories are older and use a different API, that is why kerberos libraries needs to be installed from koji: http://koji.fedoraproject.org/koji/buildinfo?buildID=410384

Installation of LDAP

OpenLDAP server can be easily installed from repository by typing `yum install openldap openldap-servers`. OpenLDAP recently switched it's 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 `systemct start slapd`. When the server is running it's 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 in Annex B, `db.ldif` file contains basic setting with new suffix, new root DN and new root password, `base.ldif` file creates root node and two sub nodes – one for users called People 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.

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`. To set up module to communicate with LDAP directory server that was configured with ldif files from Annex A and running on local machine configuration 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`.

Implementation

Overview

Because RADIUS protocol can't provide enough information for id_provider and authentication is closely connected with authorization, RADIUS provider will provide only authorization service, thus will operate only as an auth_provider. The provider is called rad as abbreviation from radius and rad is also prefix for all configuration options of this provider and majority of functions. The interface of every auth_provider consists of initialization function sssm_$provider$_auth_init where $provider$ is name of called provider, in this case rad. This function initializes provider, which means that it loads options, allocates system resources and prepares environment so that provider is ready to react to requests. Part of the environment preparation is registering function that will be called when request appears. After handling request, every provider calls function be_req_terminate that ends current request and passes results to responders.

For RADIUS client part I have finally chosen `krad` library, which is short of Kerberos RADIUS. I have not

mentioned it in RADIUS section, because it is not standalone library, but part of Kerberos project. This library provides needed functionality plus it is asynchronous library, which is big advantage. Drawbacks are, that this library is part of the Kerberos project and not standalone and therefore depends on Kerberos and 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 runtime. Another disadvantage is that this library is brand new and API and ABI are still unstable, but I was assured that ABI compatibility will be preserved. However those drawbacks were discussed and acknowledged by SSSD project leader.

Architecture

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 which 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.

Now I am going to describe purpose of functions and structures related to provider 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 and this is the function, that is called by monitor to start this provider. Structure `be_ctx` holds the context of SSSD domain with all information around it such as id_provider, auth_provider etc. Structure `bet_ops` contains function that should be called for incoming requests and last parameter is storage for internal data. This function ensures, that options will be loaded, creates provider context `rad_ctx` with those options and stores it in `pvt_auth_data`.

`struct rad_ctx`

> This structure holds provider context, because RADIUS protocol is stateless, it holds only 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 it's 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 can be 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 options.

`struct rad_options`

> This structure is accessible only within rad_init.c file and is used as a temporary storage of options

and for preventing multiple options loading. Options are loaded only once into this structure and next time are reused from this structure instead of redundant loading.

In the Illustration 5 you can see callgraph of RADIUS provider initialization with significant functions.

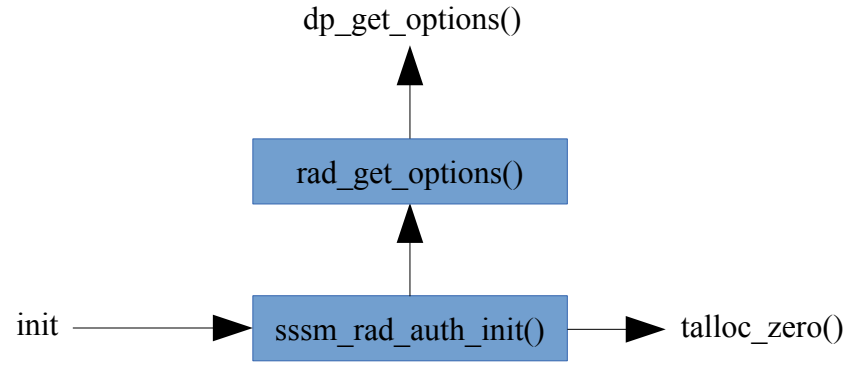


Illustration 5: Initialization callgraph

Now I will mention functions that are used to respond to authentication requests.

`void rad_auth_handler(struct be_req *be_req)`

This is the function that gets called if any request appears and be_req is structure, which contains data about this request. The only purpose of this function is to check, if it is possible to process this request and pass it to another function for processing.

`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 request depends on type of PAM command, it also checks if this type of PAM command is supported by rad provider.

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

This function assembles RADIUS packet with user data and sends it to server for authentication. It takes provider context and PAM data structure as parameters even if they are present in request so that they are easily accessible. This function creates radius request rad_req with information needed to process given request, initializes verto and kerberos context, creates RADIUS client with the use of krad library and fills it up with user data. Then it passes created client to krad library and starts verto event loop.

`struct rad_req`

This structure holds all information needed to process request by rad provider. Apart from original request it holds also variables that are needed by krad library such as kerberos and verto contexts.

`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_auth_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's purpose is to read response and act upon it – send either `PAM_OK`, or `PAM_PERM_DENIED` back to responder. Retval variable contain return value from internal krad function that receives and processes reply from server, req_pkt and rsp_pkt are structures containing request and response packets and in data part is stored radius request.

Callgraph of request processing with important functions is in Illustration 6.

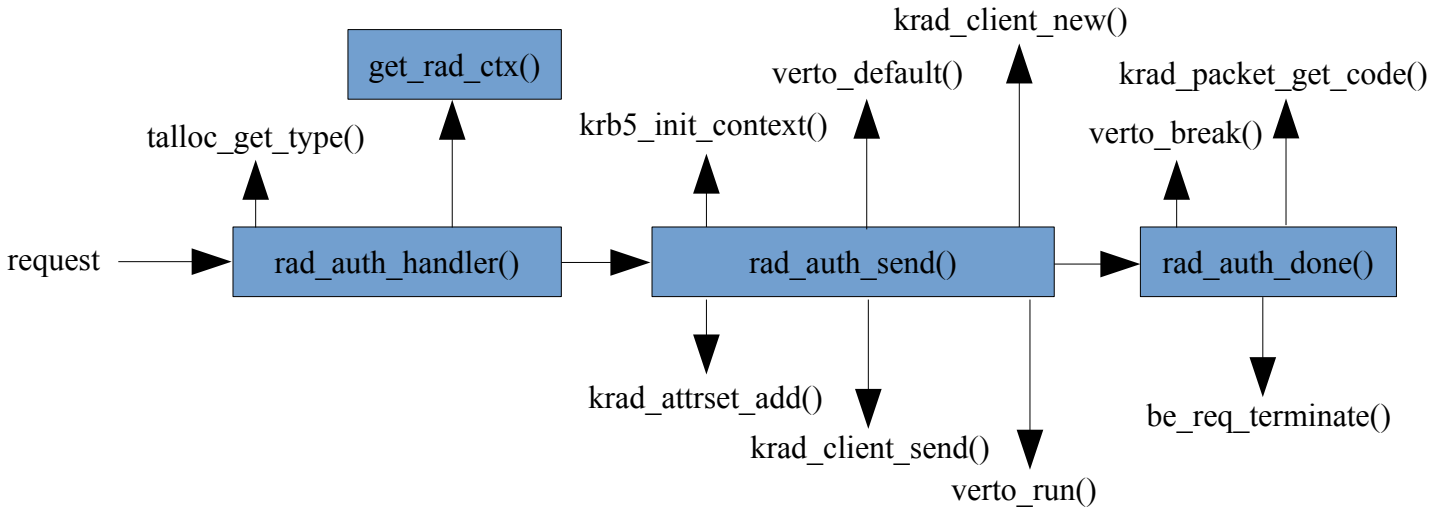


Illustration 6: Request processing callgraph

Testing

Probably some screenshots with Wireshark packets, overview of log files. I don't suppose to manage writing proper tests.

Conclusion

Good documentation is missing for every other project.

I should mention somewhere that SSSD is async.

Possible improvements for the future development – RADIUS/local user translation.

Sources

[1] Remote Authentication Dial In User Service (RADIUS). [online]. [quoted 2013-04-24]. Available from: https://tools.ietf.org/html/rfc2865

[2] HILL, Joshua. An Analysis of the RADIUS Authentication Protocol. [online]. 2001 [quoted 2013-04-24]. Available from: http://www.untruth.org/~josh/security/radius/radius-auth.html

[3] The FreeRADIUS Client Library. *FreeRADIUS* [online]. [quoted 2013-04-28]. Available from: http://freeradius.org/freeradius-client/

Annexes

Annex A

Sample SSSD configuration file (located in /etc/sssd/sssd.conf).

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

[nss]

[pam]

[domain/RAD]
id_provider = ldap
ldap_uri = ldap://127.0.0.1
ldap_search_base = dc=ondra,dc=hujnak,dc=cz

auth_provider = rad
rad_server = localhost
rad_secret = testing123

debug_level = 0x05f0
```

Annex B

LDAP ldif files used for OpenLDAP server setup.

db.ldif

```
dn: olcDatabase={2}hdb,cn=config
objectClass: olcDatabaseConfig
objectClass: olcHdbConfig
olcDatabase: {2}hdb
olcDbDirectory: /var/lib/ldap
olcSuffix: dc=ondra,dc=hujnak,dc=cz
olcRootDN: cn=admin,dc=ondra,dc=hujnak,dc=cz
olcRootPW: testing
olcDbIndex: uid pres,eq
```

```
olcDbIndex: cn,sn,mail pres,eq,approx,sub
olcDbIndex: objectClass eq
```

base.ldif

```
dn: dc=ondra,dc=hujnak,dc=cz
dc: ondra
objectClass: top
objectClass: dcObject
objectClass: organization
o: ondra.hujnak.cz

dn: ou=People,dc=ondra,dc=hujnak,dc=cz
ou: People
objectClass: top
objectClass: organizationalUnit

dn: ou=Group,dc=ondra,dc=hujnak,dc=cz
ou: Group
objectClass: top
objectClass: organizationalUnit
```