VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TIME-BASED ACCOUNT POLICIES IN FREEIPA

ČASOVÉ POLITIKY V SYSTÉMU FREEIPA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. STANISLAV LÁZNIČKA
AUTHOR

VEDOUCÍ PRÁCE Ing. PAVEL OČENÁŠEK, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá běžnými problémy časových politik, které jsou využívány v rámci procesu přihlašování uživatelů. Jsou rozebrána řešení v některých jiných současných systémech. Dále je čtenáři představen projekt pro správu identit FreeIPA, autor se zaměřuje hlavně na správu uživatelů a politiky pro jejich autorizaci. Je také představen projekt SSSD se zaměřením na jeho propojení se systémem FreeIPA. Po vytvoření návrhu řešení problému časových politik je tento návrh implementován do systémů FreeIPA a SSSD.

Abstract

This thesis deals with the common problems when implementing account policies based on time in the user authorization process. The reader is shown how this problem is solved in some of the current systems. FreeIPA identity management project architecture is presented with the focus on its user management and user authorization policies. The SSSD project is described with aim on its connection to FreeIPA. The author creates a design for time-based account policies functionality and implements it in FreeIPA and SSSD systems.

Klíčová slova

FreeIPA, SSSD, časové politiky, Active Directory, 389 Directory Server, Python, C

Keywords

FreeIPA, SSSD, time-based policies, Active Directory, 389 Directory Server, Python, C

Citace

Stanislav Láznička: Time-Based Account Policies in FreeIPA, diplomová práce, Brno, FIT VUT v Brně, 2015

Time-Based Account Policies in FreeIPA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Pavla Očenáška, Ph.D.

....
Stanislav Láznička
May 26, 2015

Poděkování

Poděkování patří mým rodičům, kteří mě podporovali po celou dobu mého studia, a také vývojářům z projektu FreeIPA, zejména panu Martinu Koškovi, za jeho trpělivost a čas, který mi věnoval.

Contents

Chapter 1

Introduction

Ever since the beginnings of the industries, there existed a problem of letting only the factory employed people in the working halls. And, of course, the foremen needed to know if all their workers came for the shift and are all doing the job they are supposed to do. Vice versa, it should not happen that a random person comes to work, pretending they are the new foreman and that all the workers have to listen to them.

The solution to such problems would seem simple at first - have someone, say, the head of the company, be sitting in that factory and overlooking the working process. But the industries grew, there were more buildings to a company, there were more people to look after. Luckily, that could still be solved by building a hierarchy of people in such companies, dividing privileges and responsibilities among these people.

The hierarchical order in companies would keep its shape throughout the whole 20th century and would be working quite well during those years. Even the coming of the computers wouldn't seem to change it much as in the very start of their era, there wouldn't be many people able to use them. The companies, however, saw the potential of the later versions of computers that didn't need a whole building just to get them to do something and with the further progress in the computer industry, these machines found their place as a main tool in many companies. The problem is that people working with computers still need to be looked after somehow, see whether they are doing their job, see whether they are not accessing what they should not be accessing and/or check whether what they do may not, even without their actual intention, do any harm to the company's computer infrastructure, for example by inexperienced use of the person's computer.

This last thing is what this thesis will mostly be about. There will be some current solutions introduced. Then, the FreeIPA identity management architecture will be presented. After that, a new design allowing time-based account policies functionality to FreeIPA will be proposed. The implementation of such design will follow. Next, a design of processing time-based policies received from FreeIPA in SSSD system will be described, followed by a description of this design implementation. The results of the implementations will be summarized in the conclusion chapter of this paper.

Chapter 2

State of the Art

In this chapter, some of the current solutions that can be used for identity and account management will be presented. With each of these solutions, the way they implement time-based account policies will be emphasized. The main goal of this chapter is only to give a basic look at how the problem of time policies is solved in some of the commercial and open-source solutions. Therefore, not all the current technologies will be mentioned.

2.1 Active Directory

This section will describe some basic concepts of the Active Directory system. The author focuses on the account policies of the system with main focus on time policies. The following description is based on the official sources[13].

Structure and Storage

Active Directory (AD) is a tool that helps to store and organize objects in a network in a hierarchical structure. This structure is called a logical structure and the objects contained in such a structure are users, computers and other physical devices. Now, a logical structure could be thought of more like a container for all those physical devices, and the actual basis of this structure are forests and domains.

A **forest** forms a security boundary of the logical structure. It can be structured so that it provides data and service autonomy and isolation. This isolation means that the physical topology covered by the logical structure is removed but can be reflected by site and group identities.

Several **domains** can be structured in a forest so that they provide data and service autonomy but they don't offer any isolation known from forests. Domains exist mainly for the reasons of replication optimization in a region. So, the main difference between a domain and a forest is that two domains in the same forest can share the same data, while sharing data between two forests is not possible.

The data in the Active Directory system is being stored as objects which are defined in the schema. Note that there is a difference between a structure object and storage object. The definitions of objects in the schema are used to ensure that the stored data is valid (e.g. the type of values of data in object is correct). This of course means that every object has to be defined in the schema before the data of such an object is stored in the directory. What is actually important for the time-based policies research of Active Directory is that all physical data is stored in a physical structure which is represented as a database that is stored on all domain controllers in a forest. How this is important will be shown later.

So, based on the previous findings, the structure and storage in Active Directory has its own architecture. This consists of four parts[13]:

1. **Domains and forests**. Alongside with organizational units (OUs), these form the core elements of the AD logical structure. A forest defines a single directory, a security boundary for this directory. Forests contain domains.

2. **Domain Name System**(DNS) support for AD. It works as a name resolution service to domain controller locations and also as a hierarchical design so that the organizational structure is easier to be understood from the naming.

3. **Schema**. As mentioned before, this is where definitions of all objects in the directory are stored.

4. **Data store**. The data store is the part that manages data on each domain controller (as mentioned before, data is stored on each domain controller in the forest).

All the information from this section is well summarized in figure 2.1.

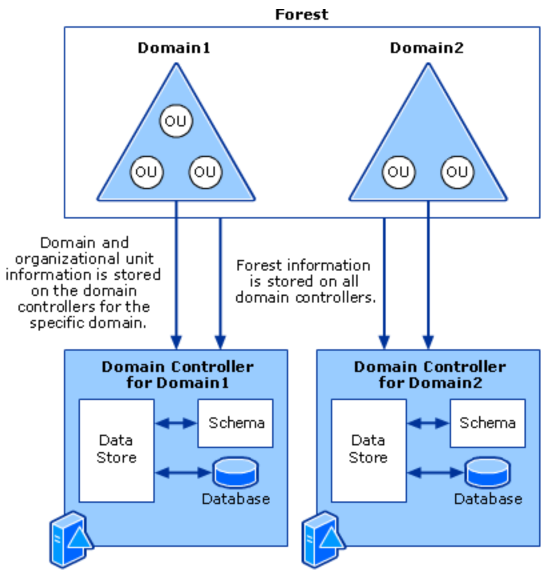

Forest
Domain1
Domain2
OU
OU
OU
OU
OU
Domain and organizational unit information is stored on the domain controllers for the specific domain.
Forest information is stored on all domain controllers.
Domain Controller for Domain1
Data Store
Schema
Database
Domain Controller for Domain2
Data Store
Schema
Database

Figure 2.1: Active Directory structure and storage architecture[13]

Account Policies

In this section, the focus will be on account policies of AD. Rather than a policy that is set right at each one specific account, an account policy will mean a policy that can somehow influence an account. Even though this may not be surprising, AD actually has its set of policies called Account Policies which will also be analyzed later in this chapter. To better understand how certain aspects of account policies work and are stored for an account, a figure A.1 in Appendix A showing a sample account object might be helpful.

Most of the policies that can be set in AD is performed through the Group Policy Objects (GPOs). These objects are special objects that can be linked to a certain domain and/or organizational units. By default, they apply to all users that are authorized access in the actual domain/organizational unit they are linked to. Although, this behavior can be changed and GPOs can be set to work only for certain Security Groups in the part of AD they are linked to (or, on the other hand, some Security Groups can be excluded from the influence of the GPO).

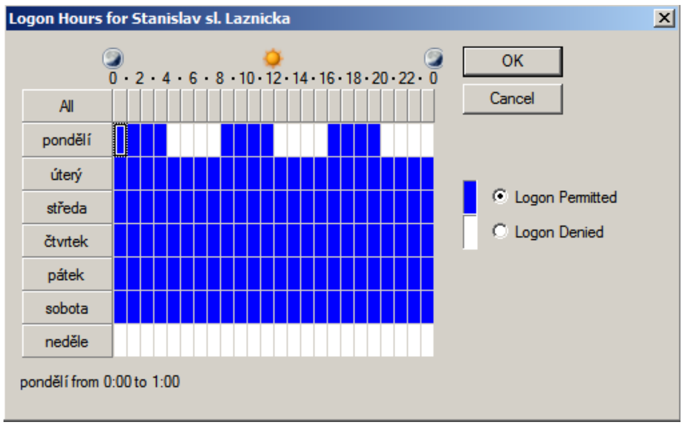
There are many things that can be set for a user account or a device using the GPOs. These things split in two groups - **Computer Configuration** and **User Configuration**.

Computer Configuration is more connected to the run of the machine that finds itself in the AD hierarchy. It allows to define which programs should be installed, allows to add scripts to be run at startup and shutdown and mostly to setup the machine itself. It also allows to set the so called Account Policies. These Account Policies consist of three main other settings - Password Policy, Account Lockout Policy and Kerberos Policy. Password Policy allows to set such things as how long might a password be used, its length, whether to encrypt passwords and so on. Account Lockout Policy defines how long should an account be locked after a certain number of invalid logon attempts. Kerberos Policy sets some values important for the Kerberos server, such as maximum lifetime for service and user tickets. A GPO containing default Account Policies is usually created with the creation of a domain and can be altered later.

As the name suggests, **User Configuration** allows the GPO to have much more impact on the user. This means that the user experience with the system is influenced. For example, one can set which software can be installed by the user, which scripts should be run on user logon/logoff, but also such things as which desktop background picture should be used.

There is another way of controlling an account aside from GPOs and that is directly at each user account. These policies include allowing Kerberos encryption support, password policies (e.g. user has to change their password on next logon, password encryption etc.), setting up account expiry date and also enabling/disabling the specific account. However, the most important setting for this thesis is the **Logon Hours**.

The Logon Hours account policy enables for an administrator to specify which hours should the user owning the specific account be able to login into the system. The figure 2.2 shows the user interface that allows changes in these settings. The logon in each hour is either permitted or denied, which is represented by blue and white colors in that particular order .



[✓ Logon Permitted] Logon Denied

pondělí from 0:00 to 1:00

Figure 2.2: User interface of the Logon Hours policy settings[12]

As with other policies set for each individual account, Logon Hours policy is stored in the certain object that the account belongs to (attribute `logonHours`). In this object, the values set in the Logon Hours dialog of the AD user interface are represented as an array of bytes. Each day of the week gets 3 Bytes, which makes 24 bits a day, each bit representing an hour in the day. The first three Bytes in the array belong to Sunday, then to Monday and so on. The value of each bit determines whether the logon is permitted in the hour or not (1 for yes, 0 for no).

A very important thing to realize is that **Logon Hours are being stored as a time record in UTC**. The time stored will get the time zone offset given by the domain controller of the domain to which the account being set belongs to. This of course means that the **local time** of the logon hours **is going to be different** in those parts of the world that **differ in time zone**.

Note that there is no such support in GPOs and the Logon Hours policy has to be specified for each user manually. The administrator of the server has the ability to run a script to enable this policy for a certain list of users, however, the user-picking for the list still has to be made somehow. To make the user log off according to the settings made for an account's Logon Hours, it is also required to have a GPO that will force the user to log off when their Logon Hours ended.

2.2 389 Directory Server

389 Directory Server is an LDAP server that is being developed by Red Hat. One of its features is the support for managing access control which, by using special rules, also supports defining access at specific time of day or day of week. It also offers plug-in architecture and among its plug-ins, there is one that allows time-based lockout policies. This plug-in will too be analyzed in this section. This section is based on the administration guide to the 389 Directory Server[10].

Access Control Management

In 389 Directory Server (389DS), the mechanism that defines user access is called access control. When the user sends BIND operation request, the server takes the information from that request and uses it to allow or deny access based on Access Control Instructions (ACIs) that are defined on the server[10]. User can be granted permissions to following operations on the directory: read, write, search and compare.

The ACIs can be applied for different parts of the directory - either the entire directory, its subtree, specific entries in the directory or for a specific set of entry attributes. It can also be applied to a specific user, user group or role, and for a location given by IP address or a DNS name.

ACIs are stored as attributes of entries in the directory and the `aci` defining attribute can be used in any entry in the directory. It consists of three parts - target, permission and bind rule. The permission and bind rule parts are set as a pair called access control rule. The permission is granted depending on the bind rule it is in pair with.

The following `aci` attribute was taken from the Administration Server object created during the 389DS installation. It allows the Server Group from the installed server to delegate permissions to read, search and compare operations in the Administration Server subdirectory.

```
aci: (targetattr=*)(targetfilter=(nsNickName=*))(version 3.0; acl "Enable
delegated access}; allow (read, search, compare) groupdn="ldap:///cn=389
Administration Server,cn=Server Group, cn=localhost.localdomain,ou=local-
domain, o=NetscapeRoot";)
```

The details of setting target and permissions are not important for this thesis, but the way bind rules work and their possibilities will be analyzed in the next section.

Bind Rules

These rules[10] are used to control access to certain operations on parts of the directory specified in the ACI within the certain bind rule. If a user wants to perform such operation, they need to **bind** to that directory. To do so, they send a bind request with their credentials (e.g. a certificate). Using the bind rules, user is granted access or not.

The bind rules define who, where from, at what time and how should access the object/directory. To do that, the bind rules use a special syntax. The possibilities of the syntax will be analyzed in this section.

There are several options in defining **who** can access the directory. It is possible to allow access of a user, of several users, and to exclude users from accessing the resource. In a similar way, access can be granted or rejected to members of certain groups or roles. There are also ways to allow access to a resource for anyone (Anonymous Access), anyone already authenticated in the directory (General Access), to entries owned by a certain user (Self Access), and to users with DN same as the parent of the targeted entry (Parent Access). It is also possible to control access based on value matching. That means that when user binds, a value of a specific attribute must match the same value of the accessed resource attribute. The specified attribute does not necessarily has to be a user name. This allows higher spectrum of use and can be also used to allow access for a specific group or role.

The 389DS allows to set **location**-based policies. This can be done either by IP address or by DNS name. When using IP addresses, rather that the usual notation, a wildcard character (*) is used to represent a subnetwork. It can also be combined with a subnetwork mask after the wildcard IP address string. While using DNS names, a fully qualified DNS domain name is highly recommended. Non-fully qualified DNS domain name would mean a potential security threat. The wildcard charater is also supported in DNS names.

There are also policies to force a user to use a certain **level of security in connections**. It is therefore possible to force a user to establish an encrypted connection for security demanding operations, such as changing password. A similar to this setting is an **authentication method** for the directory. There are four possible methods of authentication: none (default, user is not required to authenticate), simple (a user name and a password is required), SSL (some kind of PKI credentials are needed - the user should have a SSL certificate stored on some device) and SASL (user is required to bind to directory using Simple Authentication and Security Layer connection).

The last type of bind rules are the **time-based** rules. These rules allow to set time of a day and/or days of the week when a user is allowed to bind to the directory. An important thing to note here is that the **time used to evaluate these rules is the time of the directory server**. This is the same behavior as with Active Directory. It, too, means that the users in different time zones will need to try to bind with different local times in order for the bind operation to success.

While setting the time-based bind rule, there are two keywords in 389DS - `timeofday` and `dayofweek`. The syntax of setting this rule is as follows:

```
timeofday operator time
```

The `operator` can be any of =, !=, >, >=, < and <= with the semantics known from mathematics (= means equals to). The `time` argument has the form of the 24 hour clock without delimiters ('0000' to '2359'). The `dayofweek` takes comma separated string of English abbreviations of the days of the week - mon, tue, wed, thu, fri sat. An example bind rule that allows user binding from 8:00 to 17:00, Monday through Friday:

```
(timeofday >= "0800" and timeofday <= "1700" and dayofweek="mon, tue, wed,
thu, fri";)
```

It is obvious that 389DS gives the administrator more freedom in setting time-based access policies as it allows to create a rule not only for full hours, but also time in between each two full hours. The principle of working only with the server time is, however, the same.

Time-Based Account Lockout Policies

As mentioned in the first part of the 389 Directory Server section, the 389DS allows plug-ins to be deployed in the system. One of these is called the Account Policy Plug-in and allows to lock a user account based on its age and/or inactivity. It does this based on the relative time settings. This means that the policy is triggered after a certain time from a certain event - either account creation or last login time.

The Account Policy Plug-in works as follows. If it is active and is configured right for inactivity lockout, the user last login time is checked at each new login. If the inactivity of the user exceeds the amount of time set, the account is locked until the administrator unlocks it. The Account Policy Plug-in can also be used just to record last user login, though. The information about latest user login time is stored with the user account object in both cases. The way the plug-in works when the lockout should be based on the account creation date works the same with the time checking at authentication process, however, the last login time does not need to be recorded.

Chapter 3

LDAP

As shown in chapter 2, LDAP can and is used for identity management. Two solutions using LDAP were shown - Active Directory and 389 Directory Server. FreeIPA[5] uses the latter for its storage purposes, therefore it also uses LDAP. In this chapter, this technology will be briefly introduced to the reader.

LDAP is “Lightweight Directory Access Protocol, a standard, extensible Internet protocol used to access directory services”[8]. In this protocol, a client usually creates request messages and sends them to a server. The server processes the request and eventually sends response back to the client in one or more messages. LDAP is not text-based protocol like for example HTML or POP.

3.1 Operations

LDAP provides nine basic protocol operations in three categories[8]:

1. **Interrogation operations: search, compare**. These operations serve to get information about the directory.

2. **Update operations: add, delete, modify, modify DN (rename).** These operations serve to modify information in the directory.

3. **Authentication control operations: bind, unbind, abandon.**

    - **bind** operations allow a client to identify themselves to the directory by providing their authentication credentials,
    - **unbind** - terminates the session,
    - **abandon** - allows client to indicate that they no longer need the result of a request they sent before.

Note that LDAPv3 is designed to be extensible and in some implementations, there might be even more kinds of operations available.

3.2 Models

In LDAP, there are four models that describe the system[8]. The models are: **Information Model, Naming Model, Functional Model** and **Security Model**. In this section, the models will be briefly described.

Information Model

This model defines the types of data that can be stored within the directory[8]. The very basic unit of information that can be stored is an **entry**. An entry usually describes a real-world object or a certain part of a system (e.g. a person, a web page). Each of these entries have their own **distinguished name** (DN) and a set of **attributes** that describe the entry object. Each attribute has a type and one or multiple **values**.

All entries have a set of attributes that they must have so that an object can be created, and a set of attributes that they may have but that are not necessary for the object existence. Any attribute that is not required or allowed is prohibited and the entry can not have that attribute. This information about possible attributes in entry objects is stored in the so called **directory schema**.

Naming Model

This model defines how the data is organized in an LDAP directory and how to refer to certain data in the directory. According to the model, the data should be arranged in an inverted tree structure. This means that there is only one root entry which is a parent to zero or more entries which are again parents to other entries and so on. The root entry of the directory is a special LDAP entry that contains configuration information about the server and is not usually used to store information[8]. Every other node contains some data.

The naming model is built in the described way so that it can give a unique name to any entry in the directory. To refer to an entry, the distinguished names are used. The figure 3.1 shows a simple example of a part of an LDAP directory. To refer to the shaded entry, its DN would be used - `uid=bjensen, ou=people, dc=example, dc=com`. Note the inverted order of reading the name to get to the entry (right to left instead of left to right).

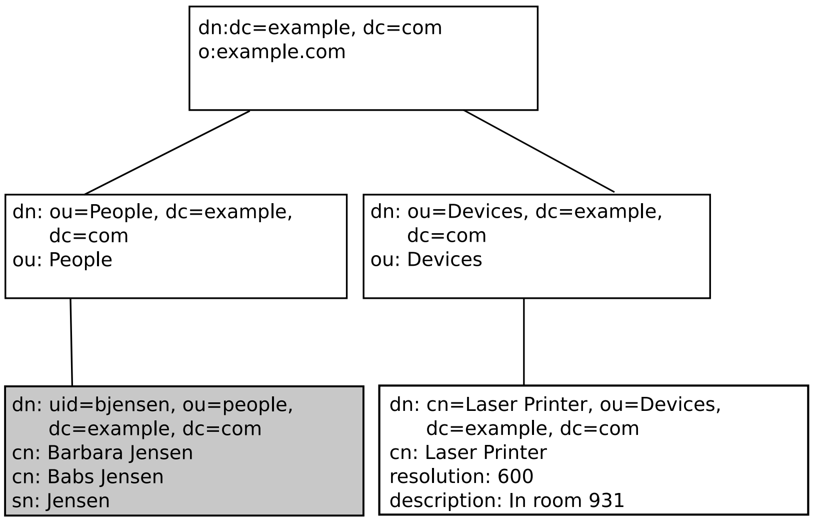

dn:dc=example, dc=com
o:example.com
dn: ou=People, dc=example, dc=com
ou: People
dn: ou=Devices, dc=example, dc=com
ou: Devices
dn: uid=bjensen, ou=people, dc=example, dc=com
cn: Barbara Jensen
cn: Babs Jensen
sn: Jensen
dn: cn=Laser Printer, ou=Devices, dc=example, dc=com
cn: Laser Printer
resolution: 600
description: In room 931

Figure 3.1: Part of Typical LDAP Directory[8]

Functional Model

The functional model describes how to access and modify the data inside an LDAP directory. It consists of operations that are divided into three groups[8]:

1. **Interrogation operations**. There are 2 operations in this group. The **search** operation allows to search and retrieve data from the directory. The **compare** operation is used to check whether certain entry has a specified attribute value.

2. **Update operations**. There are 4 update operations. The **add** and **remove** operations serve to add or remove entries in the directory. The **rename (modify DN)** operation serves to rename (modify DN) and/or move entries in the directory. The **modify** operation modifies a certain entry in the directory so that new attributes may be added, some attributes of that entry may be deleted, or an attribute value of the entry may be replaced with some other value.

3. **Authentication and control operations**. There are 2 authentication operations - **bind** and **unbind**. Aside from that, there is the **abandon** control operation. The bind operation is used to authenticate a client to the directory for as long as the connection lasts or the client reauthenticates. The unbind operation serves to end a session for a client that previously authenticated to the directory. The last operation, abandon, is issued when client no longer needs results of a certain previously initiated operation.

Security Model

The security model uses the fact that LDAP is connection-oriented protocol[8]. The privileges of a client that is authenticating to the directory may change during one connection as client may want to authenticate as different entity during one connection session. The process of authenticating to a directory is called binding - on success of a bind operation, an identity is bound to the connection between client and server. If the client does not authenticate or it does it and provide no credentials, the client is bound anonymously - with default set of privileges. Usually that is a minimal set.

Chapter 4

FreeIPA

In this chapter, the FreeIPA system will be described. Firstly, the focus will be given to each of the FreeIPA components that underlay the system. After that, the Web UI will be briefly described, as a user interface extension of time-based policies settings has to be designed later in this thesis. This chapter is based on the official website sources[5].

FreeIPA is a security information management solution for Linux with focus on the Fedora and Red Hat Enterprise Linux distributions. To achieve its purpose, it makes use of several technologies - Linux, 389 Directory Server, MIT Kerberos, NTP, DNS and Dogtag Certificate System. From the point of logical structure, it provides a server-client architecture for administration of a domain, which is usually a number of servers that require redundant data and services. The core of the system is written in Python programming language[3], the web user interface it offers is written with use of JavaScript language.

FreeIPA offers to provide management of identities (e.g. devices, users, groups), policies (e.g. host access control) and auditioning. As for Host-Based Access Control policies, the support for time-based policies that was seen in Active Directory (chapter 2.1) or 389 Directory Server (chapter 2.2) is currently missing. This thesis aims to design the support of these policies for the FreeIPA project.

4.1 Directory Server

FreeIPA uses LDAP[17] as a storage backend. For LDAP connected actions, the 389 Directory Server is being used[5]. This server serves not only for data storage, it plays the main part in the identity, authentication (along with Kerberos) and authorization services. This server also allows FreeIPA to provide multi-master possibilities, so that the data can be replicated among multiple servers.

The data in the directory is stored so that the FreeIPA objects are stored in a suffix calculated from the realm name, while certificates are stored in another suffix - o=ipaca. The storage tree with some basic objects is displayed in figure 4.1.

The access to the data is controlled using the Access Control Instructions that were described in section 2.2. That way the access can be allowed to anyone, certain users, users from a certain location etc.

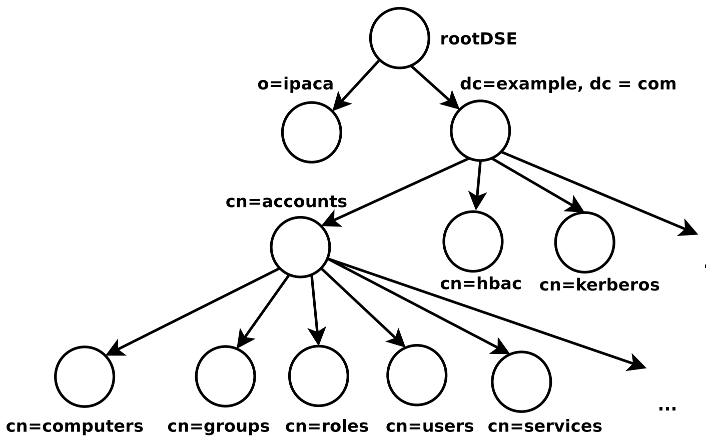


Figure 4.1: Simplified FreeIPA directory tree showing basic objects

4.2 Kerberos

The Kerberos components of FreeIPA are implemented using the MIT Kerberos software. It is used for authentication for the whole FreeIPA realm.

Kerberos[16] is a network protocol that provides means for security of authentication. For the security of authentication, no other protection of the network is required. The only requirement is that the hosts are physically secure.

The basic Kerberos protocol is based on shared secret and works like this[16]:

1. User sends request to the authentication server (AS) for the given server (Kerberos servers serve as third-party servers)

2. AS replies to the request with a message that contains so called "credentials" consisting of a "ticket" for the server the client is authorizing to and a temporary encryption key (often called "session key"). This data is encrypted with the client's key.

3. The client then decrypts the message and sends the ticket to the server. The ticket contains the user identity and a copy of the session key the user received in the credentials.

4. The session key is used to authenticate the client.

The basic Kerberos authentication process is, of course, same in FreeIPA. However, an improvement to the process is used so that a user does not have to enter their username and password all the time. In FreeIPA, the user has to authenticate and then they get what is called a Ticket Granting Ticket (TGT) from the Kerberos server. Whenever they want to login again, they present the TGT to the Kerberos server and they get the ticket for the resource they authenticate for. TGTs have an expiration time set by default. The time before expiration can be changed in the FreeIPA system.

4.3 Public Key Infrastructure

FreeIPA integrates the Public Key Infrastructure (PKI) service via the Dogtag project. It can be thought of the certificate center of the system. It signs and publishes certificates for

FreeIPA hosts and services, as well as it provides CRL and OCSP services for all software validating the published certificate.

The certificates all have limited validity. To renew these, a Certmonger daemon is introduced in the infrastructure. This daemon runs on all clients. It monitors certificates and allows to optionally refresh them.

The PKI can be configured such that it allows to be built in an existing PKI during the server installation. There are three options for PKI[5]:

- Self-signed - default PKI, uses self-signed CA certificate.
- External CA - a certificate request is sent to the CA and the PKI is chained using the certificate it gets from the CA
- CA-less - does not set up PKI server at all, only accepts signed certificates for Web Server and Directory Server components.

4.4 Domain Name System

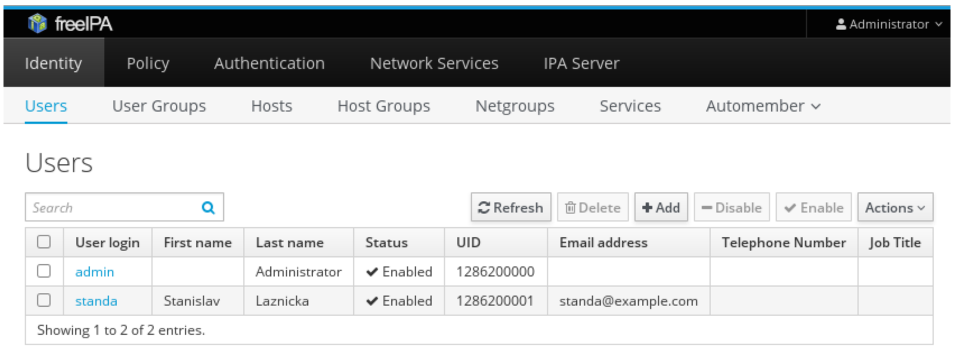
A domain name system[14] (DNS) as known from the Internet is used to create a namespace that can be used to refer to certain resources. It is designed and used as a distributed service. Its usual purpose is to translate the name from the namespace to the resource the name refers to (a URL to IP) and vice versa.

To be able to reflect DNS changes more easily and to be able to have more control of the whole system, FreeIPA allows to use its own DNS server component. It uses the BIND name server that uses the system's LDAP instance as a data backend.

4.5 Web UI and Command Line Interface

FreeIPA offers the users two supported ways of how to approach the system - the Web UI and the Command Line Interface (CLI)[5]. While the Web UI is well-arranged for the purpose of controlling the system, the CLI offers an interface either for scripting or when there is no graphical environment available for the system. Both Web UI and CLI are equal in strength.

After logging into the system via the Web UI, if the user is an administrator, they are presented with a window showing all the users (see figure 4.2).



	User login	First name	Last name	Status	UID	Email address	Telephone Number	Job Title
☐	admin		Administrator	✓ Enabled	1286200000			
☐	standa	Stanislav	Laznicka	✓ Enabled	1286200001	standa@example.com		

Showing 1 to 2 of 2 entries.

Figure 4.2: First screen after successful login[7]

The window with Host-Based Access Control Rules that is the part most important for this thesis can be found in the Policy section. The list looks the same as the list of users, which represents GUI uniformity. There are options to refresh, delete, add, disable and enable certain rules (one or more can be chosen from the list of rules). When wanting to add a rule, a window pops up with a text input to add a name of the rule. After confirming it, by clicking Add and Edit button, the window of HBAC rule edit appears (see figure A.2 in the Appendix A).

The HBAC rule edit window allows to add users, user groups, hosts and host groups affected by the rule. It also allows to add services through which the hosts should be accessed. By clicking add at either of these graphical groups, a small window pops up with the ability of moving the selected objects from Available group to Prospective (see figure 4.3).

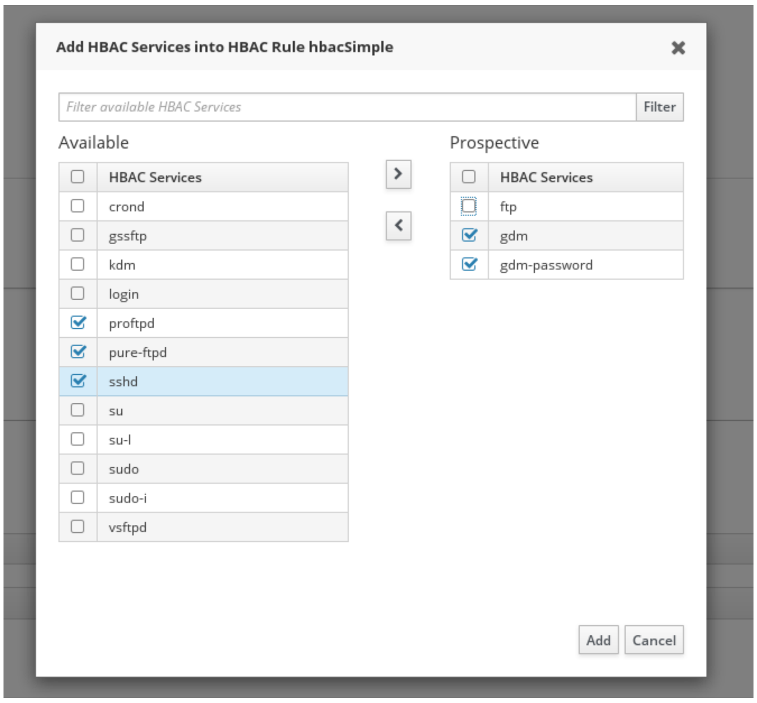


Figure 4.3: FreeIPA HBAC Add/Remove services window[7]

An example how this can be done using the CLI:

```
$ ipa hbacrule-add myRule
$ ipa hbacrule-add-user myRule --users=user
$ ipa hbacrule-add-service myRule --hbacsvcs=sshd   # adds a service to the rule
$ ipa hbacrule-add-host myRule --hosts="ipa.example.com"
```

The example shows creation of a rule with name `myRule`, which allows a user `user` to access ssh service at `ipa.example.com`.

Chapter 5

SSSD

In this chapter, the System Security Services Daemon (SSSD) will be introduced. The PAM concept will be explained as SSSD uses this concept. The way SSSD and FreeIPA cooperate will be shown.

5.1 PAM

“Linux-PAM (Pluggable Authentication Modules for Linux) is a suite of shared libraries that enable the local system administrator to choose how applications authenticate users” [15]. This means that PAM allows the developer create applications that require user authentication but that are independent on the way the user is authenticated. PAM provides a library of functions that developer can use for their application to request user authentication. As the name suggests, PAM works under the Linux operating system.

Architecture

Suppose we have an application that needs users to be authenticated before they can use it. This application is therefore designed so that it uses the standard PAM interface[15]. This interface shadows the implementation of the application itself from the authentication method and the authentication itself. The authentication method can be chosen once the application was deployed using the PAM system and its configuration.

The figure 5.1 shows the architecture of PAM. To the left, the three layers show an application called *ab*. The application has three layers - one is the application logic, second uses the PAM interface to communicate with the PAM library, and the last layer is the service it offers to a user. The Linux-PAM library in the middle gets the information about the PAM configuration for the *ab* application from the configuration file and loads the according modules. The modules are divided into four so called management groups (in the lower part of the picture). They are stacked according to their appearance in the configuration file. Some text information that is either required or offered from a user can be exchanged using the *conversation()* function[15].

Configuration

PAM services are usually configured for the whole system in files in the `/etc/pam.d/` directory or in the `/etc/pam.conf` file[15]. When the `/etc/pam.d` directory is present in

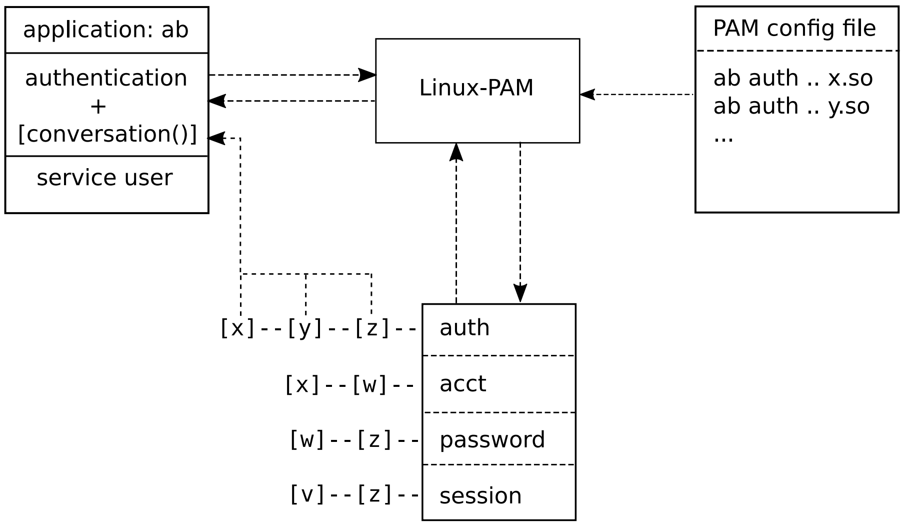


Figure 5.1: The PAM Architecture[15]

the system, the `/etc/pam.conf` file is ignored. Following is the content of configuration file `/etc/pam.d/su` for the `su` command:

```
#%PAM-1.0
auth            sufficient      pam_rootok.so
#auth           sufficient      pam_wheel.so trust use_uid
#auth           required        pam_wheel.so use_uid
auth            substack        system-auth
auth            include         postlogin
account         sufficient      pam_succeed_if.so uid = 0 use_uid quiet
account         include         system-auth
password        include         system-auth
session         include         system-auth
session         include         postlogin
session         optional        pam_xauth.so
```

Each of the configuration file lines follow the format *type control module-path module-arguments*.

The *type* is the management group that the rule corresponds to. It can have one of the following values[15]:

1. *account* - this module type performs non-authentication based account management.
2. *auth* - there are two phases of authentication in this type of module. First it authenticates the user, then it can grant group memberships or other privileges.
3. *password* - this module type updates the user's authentication token.
4. *session* - this module type serves for the things before/after user is given service.

The *control* describes what happens when the module in *module-path* fails the authentication task. It can acquire many possible values, only few will be mentioned[1]. These are:

1. *required* - failure of this PAM will lead to PAM-API returning failure, however, this failure happens only after the remaining stacked modules have been invoked.
2. *sufficient* - on succeed, if no required module failed before, immediately returns success.
3. *optional* - the result of the module matters only if this module is the only in the stack for the same service and *type*.
4. *include* - includes all lines of the given *type* in the configuration file as argument to this control.

The *module-path* is either the full or relative path to the module location. The module is given the *module-arguments* from the configuration file. These arguments are space separated tokens that may influence behavior of each module and are therefore module-specific.

5.2 SSSD

SSSD is a Linux system daemon written in the C programming language[11]. "It provides a set of daemons to manage access to remote directories and authentication mechanisms. It provides an NSS and PAM interface toward the system and a pluggable backend system to connect to multiple different account sources as well as D-Bus interface. It is also the basis to provide client auditing and policy services for projects like FreeIPA. It provides a more robust database to store local users as well as extended user data."[2]

SSSD provides possibilities of offline authentication[9]. This is performed by caching the access control rules so even when the identity management service goes offline, SSSD is still able to handle authentication requests. As a positive side-effect, the load on the identification servers is reduced as the SSSD stands in the middle of communication between the authenticated user and the identity management system. Also, it is possible to have more domains to request the identity data from.

Configuring SSSD

SSSD stores its global configuration in file `/etc/sssd/sssd.conf`. There are several sections, each contains several key/value pairs. Keys that accept multiple values have these values separated by commas. The syntax is as follows[9]:

```
[section]
# Keys with single values
key1 = value
key2 = val2

# Keys with multiple values
key10 = val10,val11
```

[1]The rest of them can be found at http://www.linux-pam.org/Linux-PAM-html/sag-configuration-file.html

The *section* usually is `sssd`, `pam`, `sudo` etc. according to the service that should be set by the key/value pairs. The section can also be name of a domain that provides identity management. The name of this domain should be prepended with `domain/`, such as `domain/example.com`, where *example.com* is the name of the domain. To have SSSD use information from this domain, the domain name needs to be added to the `domains` key in the `[sssd]` section.

SSSD and FreeIPA

One of identity providers that may be associated with SSSD is FreeIPA[9]. Based on this connection, SSSD may use FreeIPA's domain LDAP database to control access of users to some PAM services on the host.

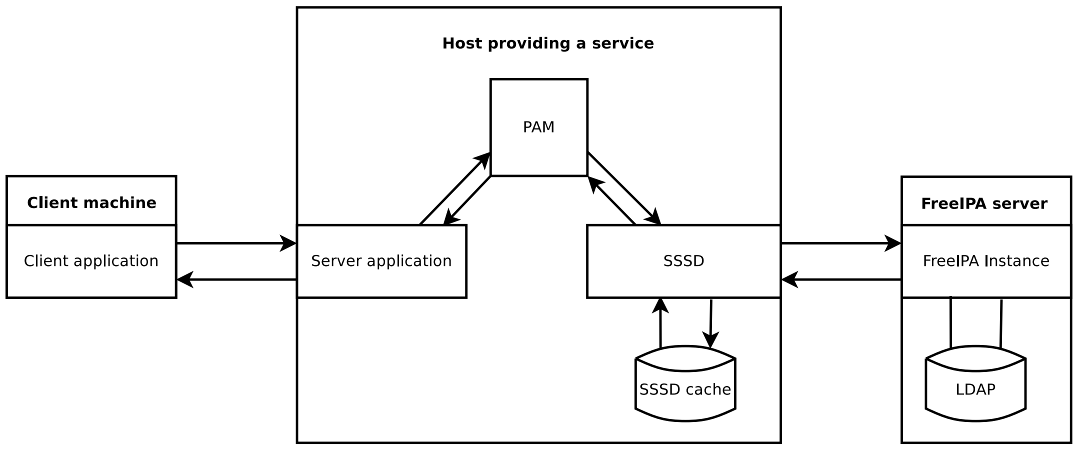


Figure 5.2: FreeIPA and SSSD

The figure 5.2 shows the architecture of FreeIPA and SSSD connection. At the start of the SSSD daemon, the daemon loads the data about the host that runs the daemon and stores them in its own cache on the host[9]. This caching is done repeatedly. When a user requests access via their client application to a host's service that uses PAM, the server instance that receives the request sends it to PAM library. The PAM library knows the request should be forwarded to SSSD. SSSD sends request for the information needed to the FreeIPA server. If the server is not online, it tries to apply a rule from its own cache. Otherwise, the information from FreeIPA server is cached and evaluated. SSSD then sends the respond via the PAM library back to the application which allows or denies access to the user based on the response.

Chapter 6

FreeIPA Time-Based Account Policies Design

In this chapter, the proposal of time-based policies for FreeIPA Host-Based Access Control Rules will be introduced. First, the possible approaches to time-based policies will be presented. Then, their possible implementation design in the FreeIPA system will be discussed.

6.1 General Views on Time-Based Policies

This section introduces some general views that might be considered while building a system that should be capable of handling time-based access rules. This is but a few possible approaches that consider time from three different levels - Coordinated Universal Time (UTC) which is everywhere the same, time local to the users and time local to the hosts.

UTC-Based Time Policies

This type of policies is the simplest one and is currently being used in both of the researched projects - Active Directory (2.1) and 389 Directory Server (2.2). Aside from the simplicity, this approach on time policies is also most secure as it does not require much additional information - no information about the time zones, no information from the client.

Here, all the responsibility for maintaining the time aspect of the access control is delegated to the server side. When an administrator sets the time of access, the time data is recalculated to the UTC time format as the server is aware about its time offset from UTC. The importance of this recalculation is in the purpose of UTC - it is the same on any place on the Earth. Therefore, local time for users from different parts of the world in different time zones will seem different, although UTC stays the same.

Example: Let us suppose an administrator sets access time at a certain day for 8:30-16:00. The server time zone is correctly set to MSK (UTC+3). Therefore, the time stored on the server would be 5:30-13:00. When a user from MSK time zone wants to access this controlled resource at 7:00 local time, he is denied access, because the UTC is 4:00. However, a user from the CET time zone (UTC+1) logging at 7:00 is granted access, as the UTC is 6:00, which is between 5:30 and 13:00.

User-Local Time Based Policies

This approach is more flexible than the one described in the previous section. However, it is also harder to maintain and more difficult in design. There are three levels of how to work with users' local time:

1. Storing the time zone for each account
2. Relying on the client's time zone settings
3. Relying only on the client's correct time.

Time Zone for Each Account

In this approach, each user has a time zone assigned in the server's database when creating an account for them and this time zone can only be changed by administrators. Then, when an administrator creates an access rule, the time that they set is the actual local time with each user. This means that when users from different time zones request access at the very same time, some might be granted it but some may not.

During the access process, when access is requested, the server checks the current UTC. It then adds the time offset for the time zone of the user requesting the access and checks whether the result satisfies the time condition that is stored with the access rule. If so, access is granted, else it's not.

There is a downside to this type of time policies. Suppose that user should be allowed access at their local time. Then, when this user is traveling, their time zones change and every time this happens, it should also be reflected in the database. That is not that hard with just one user but in a big company when there's many users traveling at a time, this can get rather complicated.

Example: The access time is set for 8:30-16:00. Users from CET (UTC+1) and MSK (UTC+3) time zones are trying to access the same resource at the very same time - user1 at 7:30 local time and user2 at 9:30 local time respectively. User2 gets the access while user1 has to wait at least another hour.

Receiving Client's Time Zone

This approach is an improvement of the previous one. Rather than storing the time zone for a user, the time zone is received from them when they want to access a resource. The time to control access to the resource would still be stored within each access rule and it would still mean the local time of any accessing user.

When a client requests access to a certain service, along with the common authentication information they also need to send the information about the time zone they are currently in (which is set on their workstation). The server is then able to calculate the local time of the user.

This method requires that the time zone on the client side is set properly. That, however, might be a problem when each user's account is not tied to a certain workstation. Therefore, the next approach to time policies is better. It too relies on proper client station setting and is more straightforward - no user's local time calculation is needed to be performed on the server..

Example: An administrator sets the possible access time to 8:30 to 16:00 as in the previous example. When a user from CET (UTC+1) time zone demands access at their local time 7:30 (no DST), they send their time zone along the access request, the host knows that UTC is 6:30 but recalculates it according to the time zone in the request to 7:30. That is however not within the 8:30-16:00 range so access is not allowed.

Local Time Only

This method is the most intuitive of all when setting the time policies or accessing a resource. It is also most demanding. Here, the administrator only sets the time users should be able to access the resource, the time represents the local time for the users. As in UTC, nothing else but the access-available time needs to be stored.

When a user sends a request for access, he also needs to add their current local time. This time is then compared on the server side with the time stored at the access rule, user is granted access if their local time is within the boundaries set by the access rule.

The shortcoming of this method is obvious - it is hard to assure that the time is set correctly on the client side. Even if it is set right, it does not necessarily mean that it will be correct all the time - the user might be in sudoers group and might change the time completely for the certain client. A user can also try to access the controlled service using their own workstation with their own settings.

Example: Access time is set for 8:30-16:00. Any user with their computer's time set between 8:30 and 16:00 would be able to access the resource.

Host-Local Time Based Policies

This approach takes a different view on the time a service should be accessed. Rather than aiming on a user, the host that offers a service is set in the main role. However, we now assume a system that involves taking care of multiple hosts, which is not a requirement to the policies mentioned earlier in this chapter.

Similar to the user-local time, host-local time policies would be divided in three groups:

1. Storing the time zone for each host

2. Storing the time zone for each access rule

3. Relying only on the host's correct time.

The trouble with the **1.** policies would be the same as in user-local time, that is multiple hosts moving across different time zones at a time, which makes it hard to keep the database integrity with reality.

The policies where the time zone is stored at access rules is different. As the access control would better be delegated to the host, there is no longer need to send any time zones from the users anywhere (hosts won't communicate between each other and each of them knows their own time zone). The time zone set at the rule does not necessarily have to be host-local, too. It can be service-local or local to a certain group of users that would need to approach the host or service. It can be therefore thought of more a rule's time zone than a host's.

When working with the host's time, the access time that is set reflects the time at the zone the host is currently in. That makes it perfect for rules where the host often changes the time zone it finds itself in. It is though not very transparent from the user's point of view and the administrator needs to be very careful because this behavior is the hardest to control as the location of all the hosts the rule applies to has to be known.

6.2 Time-Based Policies in FreeIPA

In FreeIPA, the access to certain hosts and their services is controlled via rules called Host Based Access Control Rules (HBAC rules). These rules allow the administrator to control which hosts and their services can be accessed by which users. Hosts, services and users can also be grouped and the access of the whole group can be controlled. When extending the FreeIPA access control capabilities with time-based policies, the extension must therefore extend these HBAC rules.

An HBAC rule as it is now can be represented as a tuple:

(users, user groups, hosts, host groups, services, service groups),

meaning the access is allowed to *users* (or their groups) to *hosts* (groups) and the *services* (groups) these hosts offer.

In the following text, a design of HBAC rules extension will be described.

Choosing time understanding

In the section 6.1 several approaches of time understanding were described. In this section, these approaches will be confronted with the FreeIPA system to see which can be reasonably implemented. The best possible solution from the presented will be picked.

UTC-Based Time Policies

Introducing UTC to FreeIPA is simple. The HBAC rule tuple representation has to be extended by an *accessTime* element. However, UTC does ignore Daylight Saving Time. This would make administration of the rules complicated as most time zones do count with DST. That means the rules would need to be altered twice a year just to adjust the time to these time zones (if, of course, the target time zone does have DST). That can be solved adding a DST flag to the rule. Such a rule can be then represented as follows:

(user, user groups, hosts, host groups, services, service groups, accessTime, DST).

User-Local Time Based Policies

First thing to note is that in FreeIPA, we're talking **host-based** access control rules. Bringing user-local time is therefore probably not the way the design should be heading. Indeed, two of the three previously mentioned approaches to user-local time require the user to send additional information regarding their time settings. There is not an easy way of how to enforce the users to use the correct information in the FreeIPA system. Such a solution might lead to security issues.

The only possible way of implementing user-local time to FreeIPA system would be to have the time zone of each user stored within the user account object. However, the account object does not use any object class that could easily be changed. Suppose a new object class would be created for this purpose (or, if necessary, `ipasshuser` class could be used), there would still be the problem of users traveling between different time zones. Manually managing the database integrity with the reality might then get problematic. There is a possibility of having this handled by a third party application but that would mean laying the responsibility for a part of the database to an application that does not necessarily have to meet the reliability and security standards that would normally be required from such a system.

Host-Local Time Based Policies

As mentioned above, the access rules in FreeIPA are meant to be host-based. To better understand why, this is how the system works. The access rules are created in the FreeIPA system and stored in an LDAP directory. Then, they need to be applied on the host. The rules that apply to the host, to services the host provides, or to the according groups of the host or the services, are downloaded and cached by an SSSD daemon that runs on the host. The daemon also takes care of applying the rules and therefore allows/denies access to the users trying to use the host's services.

In the section 6.1, three different host-local time policies were mentioned. The first one where the time zone of each host is stored with that host's object obviously suffers from the same problem as in storing the time zone within users' objects - it would be difficult to keep the database in sync with reality if hosts moved around the globe too much. For that reason, only the other two approaches will be mentioned - storing the time zone within the access rule object and using the host-local time.

For storing the **time zone within an access rule** object, the HBAC rule tuple would have to be expanded by both *accessTime* and *timezone*. This approach has an advantage over UTC - when using a time zone, the DST is already in the time zone information. Also, the time set is tied to a certain time zone, so it might be easier for an administrator to set the access time correctly. A HBAC rule tuple with time zone information:

(user, user groups, hosts, host groups, services, service groups, accessTime, timezone).

To successfully work with **host's local time**, the host's local time has to be correct. In FreeIPA, hosts' time is synchronized using the NTP protocol. To count the correct local time from the UTC time that has been previously set right using NTP, in Linux distributions, the `/etc/localtime` file can be used. The HBAC tuple of host-local time based policies needs the least expanding:

(user, user groups, hosts, host groups, services, service groups, accessTime).

Result

User-local time based approaches mentioned in this chapter are not suitable for the FreeIPA system. Having only the UTC approach would probably suffice although it does not provide much scalability. So on the top of the UTC approach, the method of storing a time zone with HBAC rule and method of host-local time should be used for the implementation as well.

6.3 Storing the time data

Storing time policies would mean changing the LDAP schema of FreeIPA LDAP directory. In the previous section (6.2), three of the proposed time policies approaches were chosen. To better understand HBAC rules, they were presented as tuples. However, HBAC rule has its own representation in the FreeIPA LDAP directory. It is an instance of two object classes - `ipaassociation` and `ipahbacrule` (note that `ipahbacrule` is a subclass of `ipaassociation`).

All of the chosen time policies approaches have one attribute in common - `accessTime`. Looking at the LDAP schema, the HBAC rule object already has an `accessTime` attribute from previous time-based policies implementation so luckily no change will be needed there. The syntax of this attribute's content will be described later in this section.

For the sake of time zones stored at HBAC rules, another attribute, `timezone`, needs to be added for the HBAC rule object. This attribute should contain the time zone information of the rule according to the description in the section 6.2 - **Host-Local Time Based Policies**. The time zone name string itself would be the certain time zone from the Olson database[1]. As the time zones in this database also include UTC and the DST information for each of the time zones, this helps merging the UTC-Based approach alongside the time zones withing access rules approach. Aside from that, the newly created `timezone` attribute of `ipahbacrule` object could also contain the value "`host`" meaning that the time set in `accessTime` attribute should be understood as host-local time. This means that the `timezone` attribute can be used as a "switch" between the three chosen time policies approaches.

Language of the accessTime Attribute

The `accessTime` attribute describes the time ranges for which the HBAC rule containing the attribute should be applied. The language of this attribute is inspired by the time part of the Bind Rules from the 389 Directory Server[10] although there have been some extensions and changes.

The syntax of the language is as follows:

```
keyword=time
```

The `keyword` can be one of *timeofday, dayofweek, dayofmonth, weekofmonth, monthofyear* and *year*. Each of these keywords accepts numbers from different ranges according to the semantics of that keyword (see table 6.1).

Keyword	Values range
timeofday	0000-2359
dayofweek	1-7 (1 = Monday)
dayofmonth	1-31
weekofmonth	1-6
monthofyear	1-12
year	a year

Table 6.1: Possible values of accessTime language keywords

This language includes three binary operators. One of these is the assignment operator "=", one is the range operator "-" and the last one is the concatenation operator ",". The function of the assignment operator is clear, it assigns a `time` value to the specified `keyword`.

The range operator is an operator that may only appear in the `time` part of the language. It appears between two values and it denotes a range from-to where the first operand has to be lower than the second one. Note that both range borders are included in the range.

The concatenation operator only appears in the `time` expression. It stands for a logical "or" between two values, two ranges, or a range and a value.

An example of the above:

[1]The database is also called Time Zone Database, see http://www.iana.org/time-zones

```
timezone: host
accessTime: timeofday=0800-1200,1230-1600 dayofweek=1-4,6
```

This example allows the user to access a resource on host local time between 8:00 and the noon or during the time period from 12:30 to 16:00 every day from Monday to Thursday, and on Saturday.

As the `accessTime` attribute is multi-valued, it is possible to add more than just one time policy for an HBAC rule. Between each of these policies is the logical “or” relation. Along the concatenation operator, this can be considered a replacement for the “or” logical operator from the Bind Rules. The assignments in a single expression chain replace the “and” operator. However, every keyword can appear only once per each expression (i.e. once in a any single `accessTime` value).

Exceptions in Access Time

When setting a time policy, only the time ranges when the certain HBAC rule should be applied are considered. However, it might be good to be able to also set the time ranges when the rule should not be applied even though the basic time policy says it should. These are the exceptions in access time.

A good example of when these access time exceptions should be applied are the holidays. Suppose you have a company system of hosts that should only be available during the working hours. You set the access times to, say 8:00-16:00 every day Monday through Friday. However, you don’t want the employees to be able to access the system during holidays. You need to make exceptions in your access time settings.

For the purpose of adding exceptions for the access time ranges, a new attribute needs to be introduced to the `ipahbacrule` LDAP object. The attribute is called `accessTimeExclude` because it excludes the time ranges set within it from the ranges set in `accessTime`.

The syntax of the `accessTimeExclude` is the same as for the `accessTime` attribute. `accessTimeExclude` is also multi-valued so it allows all the same features as `accessTime` does. Note that the access time exceptions have higher priority than the access times themselves. If an access time is set and an access time exception is set for the same range, the HBAC rule in that time range will still not apply.

The following is an summary example for this section:

```
timezone: America/New_York
accessTime: timeofday=0800-1200,1300-1600 dayofweek=1-3
accessTime: timeofday=1600-2300 dayofweek=3-4
accessTimeExclude: dayofmonth=4 monthofyear=7
```

In the example above, the time set is considered to be the time in the *America/New_York* time zone. When the time in that time zone is in the ranges from 8:00 to 12:00 or from 13:00 to 16:00 any day from Monday to Wednesday, the rule is applied. The rule may also apply from 16:00 to 23:00 any Wednesday and Thursday. It will not apply on the 4th of July.

6.4 User Interface Design

Designing the graphical user interface (GUI) for time policies needs to be consistent with the rest of the FreeIPA Web UI.

The figure 6.1 shows a draft of a GUI expansion for adding time policies to a HBAC rule.

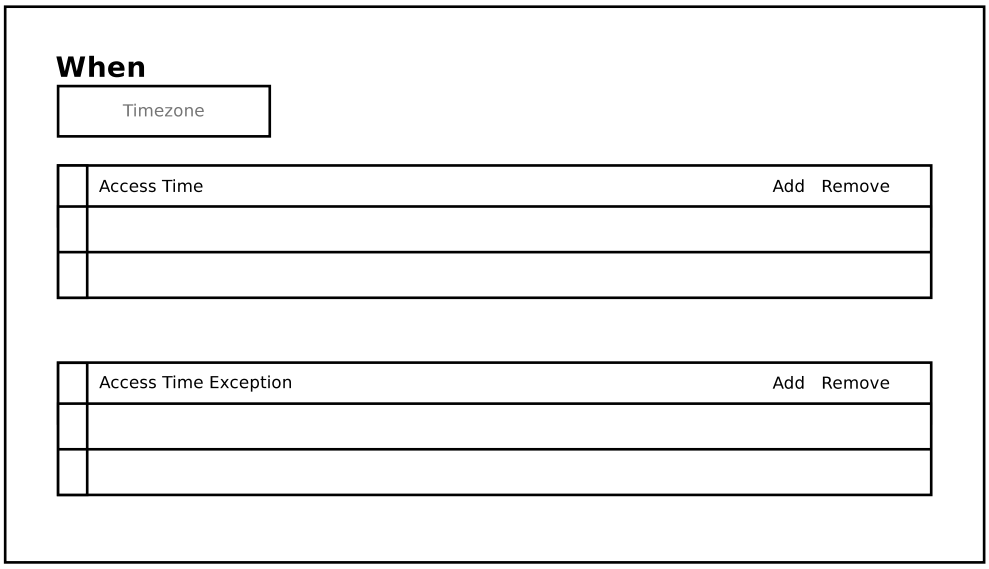


Figure 6.1: Time policies design draft

In the figure, there is the *When* title that marks that the section aims on time policies. The *Timezone* is where an HTML `select` element should be to choose one of the timezones from the Olson database and possibly the host's timezone. The tables with a header *Access Time* and *Access Time Exception* are the tables that should contain the time-based policies according to `accessTime` and `accessTimeExclude` attributes of HBAC rule LDAP object. Each row of these tables should contain a string with a time rule. There is also a possibility to add rules to the tables, or remove them by selecting them in the table and then clicking *Remove*.

After a click on the *Add* text, a dialog should pop out with a text input field to insert a time rule in. The time rule inserted in such a field needs to have the syntax described earlier in this chapter.

6.5 The CLI

The CLI interface of time policies set for an HBAC rule should be consistent with the interface of other commands that work with HBAC rule. The commands that are used for working with HBAC rules usually take the following form:

```
ipa hbacrule-do-action NAME [--argument]
```

where `NAME` is the name of the certain HBAC rule and `argument` is the argument the action works with. Both arguments are optional and may be prompted for after firing the command.

In the section 6.3, three new attributes were added to the `ipahbacrule` object in the LDAP schema. The settings of these attributes need their own commands in the CLI. Following the common interface of HBAC rules CLI commands:

```
ipa hbacrule-set-timezone NAME [--timezone]
ipa hbacrule-add-accesstime NAME [--time]
ipa hbacrule-remove-accesstime NAME [--time]
ipa hbacrule-add-exclude-accesstime NAME [--excltime]
ipa hbacrule-remove-exclude-accesstime NAME [--excltime]
```

where `NAME` is the name of the affected HBAC rule, `timezone, time` and `excltime` are the time zone name, access time ranges and access time exceptions accordingly. These last three arguments follow the time zone and access time syntax described in the section 6.3.

Chapter 7

Implementation - FreeIPA

In this chapter, the implementation of the design proposed in the chapter 6 will be described. The general principles of the FreeIPA plugin framework will be shown along with their application in extending the HBAC rules plugin with time policies.

The work that has been done as a part of this thesis is described in section 7.2 part **Extending HBAC Rule Object**, section 7.3, section 7.4 part **Changes to the Web UI** and section 7.5. The rest of this chapter describes the current state of the system that is needed to understand the changes made to it.

7.1 Plugin Framework

FreeIPA high level design was described in the chapter 4. As mentioned, the system core is written in the Python programming language. It allows building extensions on the top of it using its own plugin interface. This section is based on the document [4].

Basic Classes

The common Python programming language implementation allows a user to modify instances of objects in the runtime - changing their properties, adding own methods, removing methods of the objects and so on[4]. This behavior certainly is not appropriate for an application that aims on security. For that reason, FreeIPA system must have built its own architecture atop the basic Python programming language use.

To have a system that allows defining own functions and it keeps the object-oriented aspect of the underlying Python language, it is necessary to create basic fundaments of such a system. These are: objects, their properties, their methods and possibly functions that are not connected to any object. In FreeIPA plugin framework, these fundaments are instances of `Object`, `Property`, `Method` and `Command` classes and together with `Backend` instances form the application programming interface for FreeIPA plugins.

This altogether allows not only for the functions (`Commands`) and methods to have typed parameters, but also to lock the definitions of objects, including their properties and methods, and functions once these are defined and instantiated at the beginning of the runtime.

Parameters

As mentioned in the previous section, functions and methods in the FreeIPA plugin framework have typed parameters[4]. The parameters have to be described somehow. Therefore, all parameters are objects that inherit from a parameter base class `Param`. This of course allows uniform access to there parameters, which is needed for a uniform way of defining new commands and methods.

The `Param` class instances contain some important properties that are needed to be born in mind when creating an own parameter class and its instance. The most important[4]:

- `name` - The name of the parameter. It can be used to address the certain parameter. It may also contain some special characters that actually describe some attributes of the parameter.
- `cli_name` - The name that is used in the command line interface. It may for example be used to set a certain parameter of a function from the CLI.
- `label` - Label used in CLI or Web UI, should start with capital letter.
- `required` - Two possible values - *True* or *False*, describes whether the parameter is required or not. It is *True* by default.

The `name` property allows to set some of the other properties just by using some special characters in its value. These characters are *?, *, +*. Should they be used, only one can appear in the `name` value right after the actual name of the parameter. The *?* means that the parameter is not required and is single-valued, *** marks a non-required multivalue parameter and *+* describes required multivalue parameter. A multivalue parameter is such a parameter that can accept Python tuple of values. By default, parameters are singe-valued.

The FreeIPA plugin framework defines some basic parameters that can be used. These are for example *Bool*, *Int*, *Float*, *Str* etc.

Objects

Another mentioned elements of the plugin interface were objects. Objects are instances of the `Object` class and they contain certain properties, methods and also information about which parameters they contain. Each object also has a primary key that can be used as a reference to that object[4].

An important part of an object is the `takes_params` attribute. This attribute specifies which parameters can be passed to this object and of course the number of these parameters. As with object methods, once the system receives information of a lockdown, the object attributes and the parameters the object takes can't be changed.

Object methods are defined as Python subclasses of the `Method` class. From Python view, there is no link between FreeIPA objects and their methods. The linking is done using a naming convention. To create a method for an object of name *(name)*, the method must be named *(name)_(method)*. When a method of an object is defined, it can also take any parameter the object can take as well.

Storage Backend Methods

As operations with a database are a must for an identity management system, there is also a part of the API that offers storage methods. All these methods have `ipalib.crud.CrudBackend` abstract class as a base class. There are five basic methods that help handling the database operations[4]: `Create`, `Receive`, `Update`, `Delete` and `Search`. As FreeIPA uses LDAP as backend database, each of these methods have a special implementation just for LDAP databases - `LDAPCreate`, `LDAPReceive`, `LDAPUpdate`, `LDAPDelete` and `LDAPSearch`. As is the custom in the FreeIPA plugin framework, all of the CRUD methods and their LDAP substitutes are defined as Python classes.

LDAPObjects

Instances of `LDAPObject` class are special objects that represent objects in the LDAP database[4]. They can be referenced by their distinguished name and can represent even complex relationships of LDAP database objects.

As LDAP database objects, instances of the `LDAPObject` class need to include some special properties describing their LDAP attributes. Following are the most important:

- `container_dn` denotes the distinguished name (DN) of the container of the object entries in LDAP. It is usually populated by the environment but can also be set manually.
- `object_class` is a list of LDAP object classes associated with the object. For example, for HBAC Rule object it is `['ipaassociation', 'ipahbacrule']`.
- `uuid_attribute` defines which LDAP attribute to use for object uniqueness.
- `attribute_members` is Python dictionary. The keys of the dictionary define the name of this `LDAPObject` instance attribute, the values are lists that say which objects can be referred to by this attribute.

An example `LDAPObject` will be shown later in this chapter when part of the HBAC Rule object structure will be shown.

7.2 HBAC Rule Plugin

In the previous section, the basic parts of the FreeIPA plugin framework were covered. The information provided there is important to understand this section. In this section, the current state of the HBAC Rule plugin will be presented. The way this plugin was extended will be shown.

HBAC Rule Object

In the Python implementation of FreeIPA core, the HBAC rule object is a Python class `hbacrule` that inherits its behavior from the `LDAPObject` class[6]. The `LDAPObject` class defines some attributes that are necessary to be changed in order to create a custom class that would represent certain type of objects in the LDAP directory. Some of these attributes were mentioned in the previous section. The code 7.1 shows these attributes in the `hbacrule` implementation. Note that "..." marks there is code that is not mentioned and have no special significance.

As it can be seen in the code 7.1, the hbacrule's LDAP container object is set to api.env.container_dn, which is a function that supplies the distinguished name of the HBAC rule object container (common name "hbac"). There's also the object_class attribute that denotes the object classes of the LDAP object this Python class conforms to. The default_attributes property was not mentioned before, its value is a list of elements that should be always returned in searches. The uuid_attribute sets the "ipauniqueid" attribute of the hbacrule object to be the unique identifier of each object. The property attribute_members shows the use of this property as explained in the previous section. The memberuser attribute values here may refer to either user or group LDAP objects.

```
class hbacrule(LDAPObject):
    container_dn = api.env.container_hbac
    ...
    object_class = ['ipaassociation', 'ipahbacrule']
    default_attributes = [ 'cn', 'ipaenabledflag', ... ]
    uuid_attribute = 'ipauniqueid'
    attribute_members = {
        'memberuser': ['user', 'group'],
        ...
    }
    ...
    takes_params = (
        Str('cn',
            cli_name='name',
            label=_('Rule name'),
            primary_key=True,
        ),
        ...
    )
```

Code 7.1: HBACRule object implementation[6]

The takes_params property in code 7.1 is not actually tied to the LDAP representation of HBAC rule, it rather lists all the parameters that can be accepted by the object and its methods. The way parameters work in the implementation is shown here. The hbacrule object can accept parameter Str, which is a parameter representing a string. This parameter can be referred to by its name "cn" in any hbacrule object. As known from the previous section, the name "cn" specifies that this parameter is required and single-valued. The cli_name value "name" means that when creating a HBAC rule object, its name can be set in the command using the --name= syntax or it may be added in the interactive prompt where the command line interface will display Rule name: as stated in the label property of this parameter. This string parameter also serves as the primary key of the object.

HBAC Rule Object Methods

The HBAC rule object gets several methods. These methods are defined each as a Python class as described in the section 7.1. The methods serve mostly to change the state of the according HBAC rule LDAP object in the database. As such, they inherit their behavior from LDAP CRUD classes. The basic behavior of the LDAP backend method classes is redefined by either overriding their `execute` method or by adding certain callbacks to these LDAP method classes. A callback method is a function that can be triggered at a certain event during, before or after execution of a certain method. These callbacks are[4]:

1. `pre_callback` - called before the `execute` method,
2. `post_callback` - called after the `execute` method,
3. `exc_callback` - called when an error occurs during `execute`,
4. `interactive_callback` - called at client, allows to decide whether additional parameters should be required from a user.

The time-based policies extension for HBAC rule object will do with just the `execute` method, though.

Extending HBAC Rule Object

To extend the HBAC rule object with time-based policies behavior, new parameters have to be added to the `hbacrule`'s `takes_params` property[6]. Following the design from chapter 6, it's the `timezone`, `accessTime` and `accessTimeExclude` parameters. The implementation of these parameters will be described in the section 7.3. This change also has to be reflected in the LDAP schema, the change will be presented in the section 7.5.

As the new attributes should be also shown in search results, it is necessary to add them to the `default_attributes` property value of the `hbacrule` class. This property is an array of strings so adding the names of the attributes will do.

After the parameters have been added, new methods of the `hbacrule` class need to be defined in order to allow handling the values of the new properties.

As the name of the CLI command is deduced from the method name it stands for, the names for the methods handling these attribute values should conform to CLI design from section 6.5. Therefore, there are 5 new methods: `hbacrule_set_timezone`, `hbacrule_add_accesstime`, `hbacrule_remove_accesstime`, `hbacrule_add_exclude_access-time` and `hbacrule_remove_exclude_accesstime`. All these methods are defined according to FreeIPA plugin framework, which means they are all classes and as they need to work with the LDAP database, they inherit from `LDAPQuery` class. The behavior of each of these methods is similar so the implementation of only one will be described.

After defining the method as a class with `LDAPQuery` inheritance, the parameters of the method class need to be set. Our methods always take only one parameter, this need to be added to the `takes_options` tuple. Then, the `execute` method of this method class needs to be overridden. The `execute` method is defined as follows[6]:

```
def execute(self, cn, **options)
```

where `cn` is the LDAP common name of the object invoking the method and `options` are the options defined by `takes_options` property. In our case, what we always need is to get the LDAP distinguished name of the method invoking object (that is done using its CN from the method arguments) and then get the object's certain attribute that is handled. The changes on the attribute are performed and the system updates the object in the LDAP database.

Once the method class is prepared, it needs to be registered in the environment. That is performed using the `@register` Python decorator.

7.3 New Parameters

In the section 7.2, the need for new parameter classes is mentioned several times. The new parameters will be described in this section. These new parameters are the `TimeZone` and the `AccessTime`. The `accessTimeExclude` attribute of the HBAC rule object does not need a special new parameter class because it follows the same rules as the `accessTime` attribute.

TimeZone

In FreeIPA plugin framework, each parameter is a class that describes a parameter that can be passed either to an object or to a method. These parameters' base class is an abstract class `Param`. The parameters follow some rules described in the section 7.1.

The timezone, according to the design in chapter 6, is a string that describes a certain time zone, e.g. "Europe/Prague". Therefore, it is a string that has the restriction that it has to be a an Olson database timezone name or it may also contain the value "host" which means a host's timezone.

To satisfy the needs a timezone parameter desires, a new parameter class `TimeZone` is created. This class inherits its behavior from the `Str` class, which is a class of string parameters. The `TimeZone` class overrides method `_rule_required`. This method represents an integrity check when a parameter is passed to either a function or an object. The method uses **`pytz`** library to decide whether the value passed to the `_rule_required` method is a time zone name or not. If it is not and it is neither "host" string, it raises a `ValidationError` exception.

AccessTime

The `AccessTime` parameter represents a string that has special syntax. This syntax is the syntax of time-based policies and was described in the section 6.3. The string in this parameter therefore needs to be checked for syntax and value errors before it can be passed to an object or a function. Also, for the sake of parsing and storage, the parameter should offer means to create a normal form of the time policy. The normal form should not include any unnecessary white spaces but the keywords each with its value should be divided from other keywords by a white space.

As a string parameter, `AccessTime` parameter class should inherit from the `Str` parameter class, same as `TimeZone` class. As the syntax check is required, the `_rule_required` method needs to be redefined as well.

In the beginning of the rule syntax check in the `_rule_required` method, the input value is transformed into a normal form. This ensures that each keyword with its value is divided from other keywords and their values by a white space. It also makes sure that there are no extra spaces around the "=" and possible "-" and "," signs. Then, the input in normal form is split by spaces so each element can be checked separately. In the separate check, first the keyword of each part of the original input in normal form is checked for correspondence with known keywords. If it does not correspond, `ValueError` exception is raised. After this check, the check for correct values in the keyword argument is performed. This means, for example, checking whether the time is in HHMM format and that there is no more than 23 hours or 59 minutes set. If the check fails, `ValidationError` exception is raised. If all checks are ok, the initial value can be passed to a function or an object, else exception must be handled.

7.4 Web UI

The Web UI of FreeIPA system is a JavaScript single-page application that is formed by three types of elements - **facets**, **widgets** and **fields**[6].

Widgets are display and logical elements of the Web UI. They are JavaScript objects that usually hold a representation of a certain element in HTML. They are also able to handle the data they receive from fields and by using RPC they can communicate with the FreeIPA server to modify the data on the server.

Fields are the data elements of the Web UI. They are able to communicate with FreeIPA server using RPC and receive the data from the server as results of these procedure calls. Each field is usually connected to some widget that displays the data the field receives..

Facets represent the web pages of the Web UI. Facets are build from multiple fields connected to widgets. Facets contain specifications of the fields and widgets that form a single web page. These specifications usually say how the fields and widgets are connected and can also modify some attributes of the widgets. For example, a facet specification can modify widget's labels.

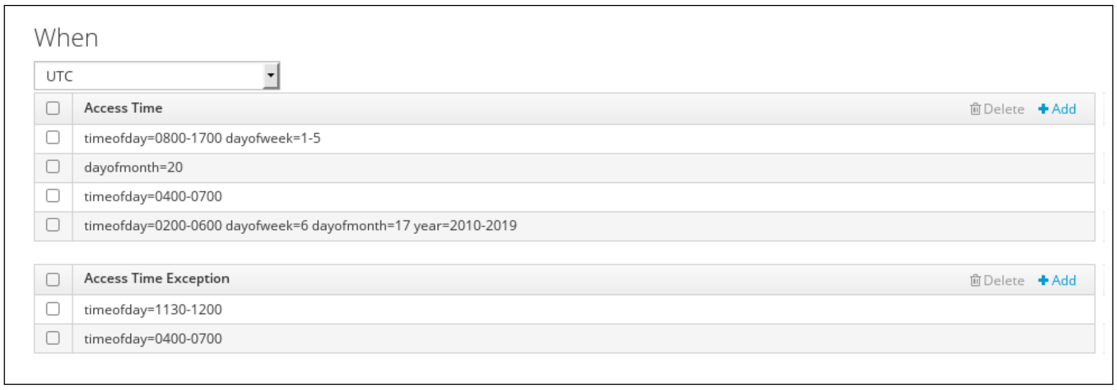
Changes to the Web UI

The changes that had to be made were done to `add_hbacrule_details_facet_widgets` function that defines specification for displaying HBAC rules in `hbacrule_details_facet` facet.

For displaying and handling the timezones, the `select` widget was chosen. This widget represents a single HTML `select` element. It offers an attribute `options` that was used to add names of time zones from Olson database to the `select` widget.

To display and modify time rules of time-based policies, a new widget and field classes were created. The field type is based on the `IPA.field` class. It provides data in a field of objects so that this data can be displayed in a table.

The new widget is based on the `IPA.attribute_table_widget`. Functions to refresh the displayed table on remove and add of a time rule had to be changed. Also, as the removal methods of HBAC rule objects from the section 7.2 can only handle one time rule sent to them at a time and a table allows selecting more time rules at once, a `create_remove_method` method needed to be created in this widget class to handle this behavior. The result of this implementation is shown in figure 7.1.


When

UTC

☐	Access Time
☐	timeofday=0800-1700 dayofweek=1-5
☐	dayofmonth=20
☐	timeofday=0400-0700
☐	timeofday=0200-0600 dayofweek=6 dayofmonth=17 year=2010-2019

☐	Access Time Exception
☐	timeofday=1130-1200
☐	timeofday=0400-0700

Figure 7.1: The time-based policies display in Web UI

7.5 Changes to the FreeIPA LDAP Schema

All the changes in this chapter also need to be reflected in the FreeIPA LDAP schema. FreeIPA extends the common LDAP schema with many of its own attribute types and object classes. The schema is stored in the `.ldif` files along the source codes of the FreeIPA system.

Most of the attribute types and object classes that are connected to HBAC rules are stored in the `60basev2.ldif` file. The definition of the `ipaHBACRule` objectClass as is reads as follows[6]:

```
objectClasses: (2.16.840.1.113730.3.8.4.7 NAME 'ipaHBACRule'
SUP ipaAssociation STRUCTURAL MUST accessRuleType
MAY ( sourceHost $ sourceHostCategory $ serviceCategory
$ memberService $ externalHost $ accessTime )
X-ORIGIN 'IPA v2' )
```

The `accessTime` attribute is already there from previous implementations. The `timezone` and the `accessTimeExclude` attributes need to be added among the `MAY` attributes, as these are not compulsory.

The attribute types for `timezone` and `accessTimeExclude` also needed to be defined. Along with the existing `accessTime` attribute types, these go as follows:

```
attributeTypes: (2.16.840.1.113730.3.8.3.14 NAME 'accessTime'
DESC 'Access time' EQUALITY caseIgnoreMatch
ORDERING caseIgnoreOrderingMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 X-ORIGIN 'IPA v2' )

attributeTypes (2.16.840.1.113730.3.8.3.18 NAME
'accessTimeExclude' DESC 'Access time - exclude these values'
EQUALITY caseIgnoreMatch ORDERING caseIgnoreOrderingMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 X-ORIGIN 'IPA v4' )

attributeTypes (2.16.840.1.113730.3.8.3.19 NAME 'timezone'
DESC 'Olson's database timezone name'
EQUALITY caseIgnoreMatch ORDERING caseIgnoreOrderingMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE
X-ORIGIN 'IPA v4' )
```

As it can be seen, all the new attribute types ignore case in either searches and ordering - there is no need to push the case of the letters. Also, the `accessTime` and `accessTimeExclude` attribute types are multi-valued as discussed in the design (6). Also, the SYNTAX of these attribute types is set for *1.3.6.1.4.1.1466.115.121.1.15* which in LDAPv3 denotes a string in the UTF-8 or ISO 10646 forms.

Chapter 8

SSSD Time-Based Account Policies Design

As explained in the chapter 5, SSSD can play the client role to FreeIPA instance. It runs on a certain host in a FreeIPA domain and caches the rules that include the information about the host access[9]. Based on these rules, it is able to allow or deny user's access to a PAM service that is run on the host.

In this design chapter, a proposal of dealing with time policies on SSSD side will be made.

8.1 Cache

The SSSD daemon repeatedly checks for rules that include the host that runs the daemon[9]. It receives a series of HBAC rule LDAP objects (we are talking only host-based access here). It then stores the rule in its own cache on the host. The cache is an LDAP database and can be accessed using common LDAP access tools. The cached record looks like this[18]:

```
dn: ...
accessRuleType: allow
cn: examplerule
ipaenabledflag: TRUE
ipauniqueid: 9431da6c-f4a4-11e4-a9c4-5254006c6e77
memberHost: fqdn=ipa2.example.com,cn=computers,cn=accounts,dc=example,dc=com
memberService: cn=sshd,cn=hbacservices,cn=hbac,dc=example,dc=com
memberUser: uid=standa,cn=users,cn=accounts,dc=example,dc=com
objectclass: ipaassociation
objectclass: ipahbacrule
originalDN: ipaUniqueID=9431da6c-f4a4-11e4-a9c4-5254006c6e77,cn=hbac,dc=exampl
 e,dc=com
distinguishedName: ...
```

It is clear from the example that the cached rule is almost the same as the rule in the FreeIPA LDAP instance. The only thing that changes is the `distinguishedName` and `dn` attribute values which need to be changed to keep the LDAP database integrity of the cache.

The change that will have to be made is that the cached HBAC rule objects should also contain the information about the time policies. Therefore, three new attributes will need to be added to the object - `timezone`, `accessTime` and `accessTimeExclude`.

8.2 Time Policies

Lets' briefly repeat what are the time-based policies that need to be dealt with. A single time-based policy according to section 6.2 can be represented as a tuple *(timezone, accessTime, accessTimeExclude)*. A *timezone* is a string. The value can be either "host" to mark host local time zone, or a time zone name according to the Olson database. Both *accessTime* and *accessTimeExclude* are strings or a number of strings that follow special syntax and describe a time range.

Parsing

The time policies are build on the FreeIPA side. They describe a certain time range. On the SSSD side, these policies need to be evaluated. It is therefore necessary to convert this description into machine representation, that is, the string needs to be converted into specific values.

The language of the *accessTime* attributes is described in the section 6.2. It is obviously a regular language.

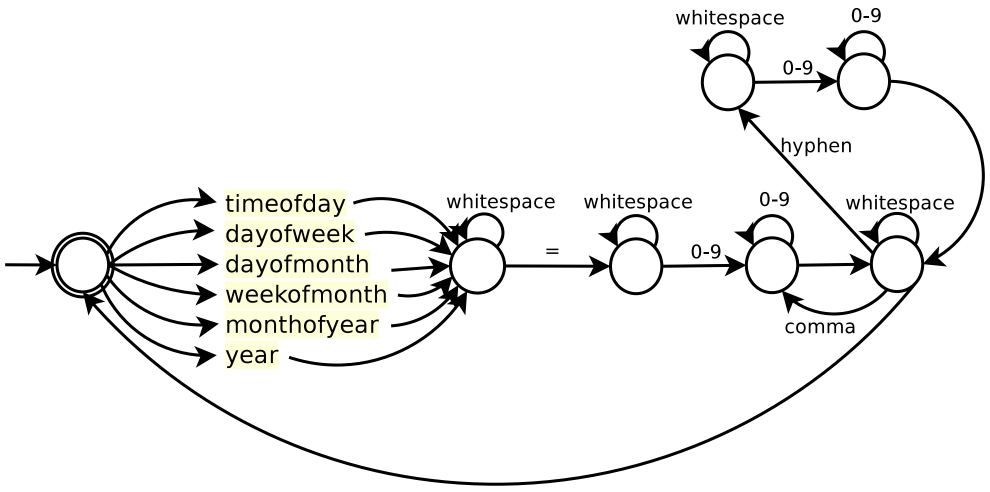


Figure 8.1: Simplified automaton describing language of time policies

The figure 8.1 shows a simplified version of automaton that needs to be implemented in case to parse the language of *accessTime* attributes. The groups of strings `timeofday, dayofweek` etc. mark the states that would need to be gone through to process the according string. The "whitespace" description of a transition marks a single whitespace character, "comma" marks the comma character and "hyphen" the hyphen character. Transitions without description are transitions with empty string. The start state is also the finish state. The automaton also supposes that the time of the policy does not necessary have to be in the normal form. That is why the whitespace reading states are also included.

In the implementation of the automaton, the application would also need to know which of the strings from the group `timeofday, dayofweek` etc. was read to determine how many numbers should be read. That would however render many extra states of the automaton and it would not be so easy to display.

Time to Different Time Zone

Other than parsing time values, the time zone settings must also be reflected into the rule evaluation. The section 6.2 mentions possibility of using `/etc/localtime` to enforce host-local time policies. The `/etc/localtime` file in Linux systems is a symbolic link to a file usually in `/usr/share/zoneinfo/` or `/usr/lib/zoneinfo`[1]. These directories contain files in the `tzfile` format. This format is used by the C function `tzset()` which can be used to initialize structures that help determine the local time. The `tzset()` function first reads the `TZ` environment variable and if it is not set, it looks for the `/etc/localtime` file which is then used to initialize the C structures that help determine the system's clock. The `TZ` environment variable can be set to point to the files in the `/usr/share/zoneinfo` or `/usr/lib/zoneinfo`. Which directory with `tzfile` format files to choose depends on the (g)libc version.

Chapter 9

Implementation - SSSD

In this chapter, the implementation of the design from the chapter 8 will be introduced. The current state of the SSSD system will be shown. Later, the changes to that system will be presented.

9.1 Current State

The SSSD system works as described in chapter 5. The SSSD gets information from the PAM library and needs to process it. One possible way of processing this information is by using the IPA provider which can evaluate certain information using the HBAC rules set on the FreeIPA side. This process of evaluating HBAC rules will be described from the point of implementation.

The figure 9.1 shows the basic control flow of the IPA provider in the SSSD when working with the HBAC rules.

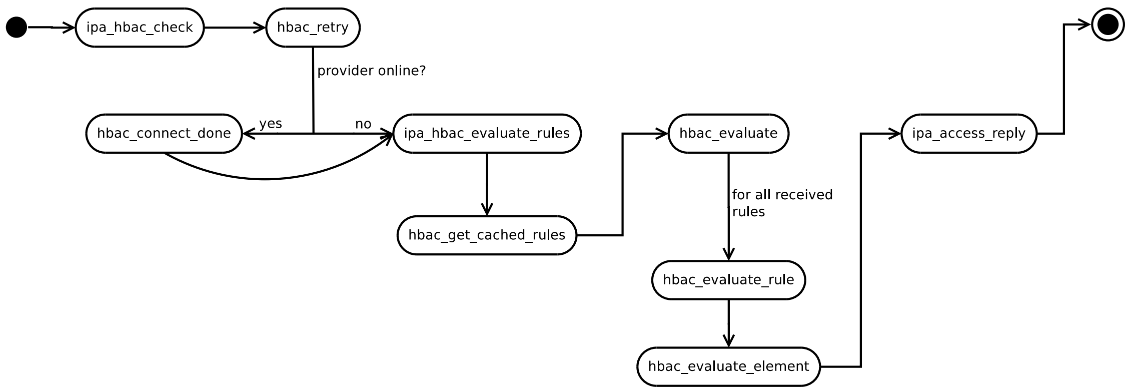


Figure 9.1: Basic control flow in SSSD IPA provider[18]

The flow in the figure 9.1 begins with the `ipa_hbac_check` function. It might be considered an entry function when working with the HBAC rules, although it is preceded by a handler function that runs some checks and then passes the control to the actual `ipa_hbac_check` function.

ipa_hbac_check

The ipa_hbac_check processes the request object it received from previous calls. First, it checks whether the account requesting access was locked or not. If so, no more actions are needed and the request will get a denying response. If the account was not locked, a structure that represents context for evaluating HBAC rules (hbac_ctx) is initialized from the request and hbac_retry is called, passing this context to it.

hbac_retry

The hbac_retry function first checks whether the connection to the data provider (FreeIPA) is online. If it is not, it just calls the function ipa_hbac_evaluate_rules. The same happens even when the provider is online but the last update of the cache was so recent that the cache refresh interval is not over yet. Otherwise a request to refresh the cache is dispatched and when that is done, hbac_connect_done is called.

hbac_connect_done

hbac_connect_done creates a new HBAC context structure from the previous request to update cache. With this structure, the ipa_hbac_evaluate_rules function is called. Also, once the rules have been evaluated, the hbac_get_host_info_step function is called which eventually, by subsequent calls, gets to the hbac_sysdb_save function that handles caching the rules from the HBAC context created in the beginning of hbac_connect_done.

ipa_hbac_evaluate_rules

This function prepares the HBAC rules to be evaluated against the HBAC context structure. It first gets all the cached rules and stores them in the HBAC context structure. Then, this structure is sent to the hbac_ctx_to_rules which transforms the rules from their cached structure to a structure more suitable for evaluation. The following is that evaluation-friendly structure[18]:

```
struct hbac_rule {
    const char *name;
    bool enabled;

    struct hbac_rule_element *services;
    struct hbac_rule_element *users;

    struct hbac_rule_element *targethosts;
    struct hbac_rule_element *srchosts;

    /**
     * For future use
     */
    struct hbac_time_rules *timerules;
};
```

name is the name of the rule, enabled is a flag that says whether this rule should be taken in account. services, users and targethosts are structures of hbac_rule_element type that represent the services, users and target hosts that are being included in this rule. The srchosts is a deprecated option for the HBAC rules. Obviously, time rules were to be once implemented, that is what timerules attribute of that structure stands for. The hbac_time_rules structure is not yet defined.

Aside from creating the rules to evaluate with, hbac_ctx_to_rules also uses the hbac_ctx structure to create an evaluation request with. This request element has similar structure as the hbac_rule structure. It describes which user requires access to which service on the host. During the creation of this request, the current time is also added among its attributes.

Once the rules have been converted from the HBAC context representation to the hbac_rule structure, an array of these rules along with the evaluation request is sent to the hbac_evaluate function. Depending on the result of that function, the access is either allowed or denied. The decision is reported using the ipa_access_reply function.

hbac_evaluate

hbac_evaluate takes all the rules from the array of the hbac_rule structure that was sent to it. It evaluates them one by one by sending them to the hbac_evaluate_rule function. It also provides it with the evaluation request structure whose attributes are being compared against in the hbac_evaluate_rule function. That function uses the unified form of the hbac_rule_element to evaluate the elements of the hbac_rule structure in the hbac_evaluate_element function.

When hbac_evaluate receives a single match from the hbac_evaluate_rule, it then sends this result to ipa_hbac_evaluate_rules. If no rules match, denying result is sent to the same function.

9.2 Extending SSSD

According to the previous section, there are certain aspects of SSSD that need to be extended so that time-based policies could be added to FreeIPA HBAC rule evaluation in SSSD. Basically, there are three things to modify - caching the time policies within each HBAC rule object, retrieving the time policies and adding them to the hbac_rule object in hbac_ctx_to_rules function, and evaluating the time rules.

Caching Time Policies

As mentioned in the previous section 9.1, caching of the rules is performed in the hbac_sysdb-_save function. As an HBAC rule is downloaded from FreeIPA, there is not much to do. As hbac_sysdb_save calls ipa_hbac_rule_info_recv, which acquires the data that is cached later on, the only thing that needs to be modified is the ipa_hbac_rule_info_send function. The attributes of an HBAC rule LDAP object that need to be stored are described here. These just needs to be expanded with the strings for the timezone, accessTime and accessTimeExclude attributes.

Retrieving Time Policies

The cached HBAC rule objects are retrieved in the hbac_get_cached_rules. Similarly to caching time policies, the attributes that should be retrieved just need to be added into an array of strings.

The time policies are now a part of the `rules` property of an `hbac_ctx` structure. They need to be added in the `hbac_rule` structure so that they can be evaluated. As mentioned before, this happens in the `hbac_ctx_to_rules` function. In this function, the attributes like users and services are parsed from the `hbac_ctx rules` property into new structure - `hbac_rule`. This structure was shown in section 9.1. The attributes are represented as `hbac_rule_element` properties of the `hbac_rule` structure. The property has the following form[18]:

```
struct hbac_rule_element {
    uint32_t category;
    const char **names;
    const char **groups;
};
```

This structure cannot be used for the time policies. Although time policies consist of two arrays of strings that describe `accessTime` and `accessTimeExclude`, they also need another string property to store the `timezone` in. Therefore, a new type of structure needs to be created:

```
struct hbac_time_rules {
    const char *timezone;
    const char **accesstimes;
    const char **exceptions;
};
```

As shown earlier, the `hbac_rule` structure already has a property of `struct hbac_time-_rules` so it does not have to be changed. The `hbac_time_rules` structure is then filled in the `hbac_ctx_to_rules` function and time policies are ready to be evaluated.

Evaluating Time Policies

The HBAC rules are evaluated in the `hbac_evaluate_rule` function. Here, the elements of the rule, such as users, services and targethosts are compared to the access request that has been received previously. The time policies must also be evaluated here.

The evaluation starts with the call of `hbac_evaluate_time_rules`. This function receives a `hbac_time_rules` structure, the time of the access request, and a pointer to set whether the rule matched or not.

The first thing to do is to decide which time should be used. The time zone might not have been set as well. In that case and in the case of the time zone to be set as "UTC" string, the time that should be used is UTC. That means that UTC is the **default** time understanding. If the time is set to "host", local time of the host should be used. Local time could be received using the `localtime` function. This function gets the local time settings from either the "TZ" environment variable or, if not set, which is usual, from the `/etc/localtime` file.

If time zone is none of the above, it is probably a name of a time zone from the Olson database. In that case, the content of the "TZ" environment variable is stored, the variable is set to the value of the time zone in the rule and `localtime` is called. Then the "TZ" variable content is restored.

Then, the content of the HBAC rule object attributes `accessTime` and `accessTimeExclu-de` is evaluated with the time received in the previous step. The exceptions in access time need to be evaluated first. Therefore, if any of the `accessTimeExclude` policies match, the pointer of match is set to false and the evaluation ends. If that is not the case, the evaluation can continue with the content of the `accessTime` attribute. The process is similar, only if a rule matches, the match is set to true.

The evaluation of each of the time rules either in `accessTime` or `accessTimeExclude` is performed in the `eval_time_rule` function. This function parses the strings of the rules and controls whether the values set in those strings are correct (for example, `dayofweek` is not set lesser than 1 or greater than 7) and also decides whether the time of the original access request lies in the boundaries set by the time rules.

If everything goes well, `hbac_evaluate_time_rules` obtains either a match or non-match from `eval_time_rule` and sends it back to `hbac_evaluate_rule` as well as returning EOK constant. In case of an error, ENOMATCH is returned.

Chapter 10

Conclusion

This thesis outlined some current technologies that implement identity management and account-based access control - the projects Active Directory and 389 Directory Server were presented. Then the FreeIPA and SSSD systems were introduced to the reader. Based on the knowledge of these two systems, their extensions to support time-based account policies were designed and implemented.

The implemented solution allows administrators to define time ranges a client is allowed access to a service on a certain host that is a part of a FreeIPA system. It offers more flexible time-based access than the two projects that were shown in the 2 chapter. The exceptions in time-based policies allow to describe time ranges when the access control rule should not apply. Having these exceptions in one place rather than scattered throughout the time-based policies might prove useful for the administering personnel as simple settings is important for security applications.

The design implemented in this thesis was discussed with experienced developers from the FreeIPA project. The current state of the implementation is not yet to be found in FreeIPA and SSSD projects upstream repositories, its deployment will require further discussion with the developers. The solution, however, is fully functional.

There are several ways in which the solution proposed and implemented during this thesis might be extended. One way would be to create a converter from a standardized time format, such as iCalendar, to the format of time-based policies in this thesis. This would allow setting the time-based policies by a third party application. The graphical interface for setting time-based policies in the FreeIPA Web UI could also be improved so that it enables a rather graphical representation of the time-based policies.

Bibliography

[1] *Linux man pages: tzset(3)* [online].[cit 2015-05-17]. <`http://linux.die.net/man/3/tzset`>.

[2] SSSD. 2015. In: *SSSD man pages* [online].[cit. 2015-05-09]. <`https://jhrozek.fedorapeople.org/sssd/1.12.4/man/sssd.8.html`>.

[3] BEAZLEY, David M. and Brian K. JONES. *Python cookbook. 3rd ed.* O'Reilly, 2013. ISBN 978-1-449-34037-7.

[4] BOKOVOY, Alexander. *Extending FreeIPA* [online]. <`https://abbra.fedorapeople.org/freeipa-extensibility.pdf`>, 2011. [cit. 2015-05-06].

[5] FreeIPA. *FreeIPA* [online].[cit. 2015-01-18]. <`http://www.freeipa.org`>.

[6] FreeIPA. *FreeIPA source code* [software].[cit. 2015-05-17]. <`https://git.fedorahosted.org/cgit/freeipa.git`>.

[7] FreeIPA. *FreeIPA Web UI* [software].[cit. 2015-01-18]. <`http://www.freeipa.org`>.

[8] HOWES, Timothy A. *Understanding and Deploying LDAP.* Addison-Wesley, 2003. ISBN 06-723-2316-8.

[9] HRADÍLEK, Jaromír, Douglas SILAS, Martin PRPIČ, et al. *System Administrator's Guide: Deployment, Configuration, and Administration of Fedora 17* [online]. [cit. 2015-05-09]. 1.
<`http://docs.fedoraproject.org/en-US/Fedora/17/html/System_Administra-tors_Guide/`>,
2012.

[10] Red Hat Inc. *Red Hat Directory Server 9: Administration Guide* [online]. <`https://access.redhat.com/documentation/en-US/Red_Hat_Directory_Server /9.0/html/Administration_Guide/`>, 2010 [cit. 2015-01-02].

[11] KERNIGHAN, Brian W. and Dennis M. RITCHIE. *The C programming language. 2nd ed.* Prentice Hall, 1988. ISBN 01-311-0362-8.

[12] Microsoft. *Windows Server 2008 R2* [software]. <`http://www.microsoft.com/en-us/download /details.aspx?id=16572`>.

[13] Microsoft. *Technet: Active Directory Collection* [online]. <`http://technet.microsoft.com/en-us/library/cc780036%28v=ws.10%29.aspx`>, 2014 [cit. 2014-12-30].

[14] MOCKAPETRIS, Paul V. *Domain names - concepts and facilities*, RFC 1034. <`http://www.rfc-editor.org/info/rfc1035`>, November 1987.

[15] MORGAN, Andrew G. and Thorsten KUKUK. *The Linux-PAM System Administrators' Guide* [online]. [cit. 2015-05-08]. <`http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_SAG.html`>, 2010.

[16] NEUMAN, Clifford B., Tom YU, Sam HARTMAN, and Kenneth RAEBURN. *The Kerberos Network Authentication Service (V5)*, RFC 4120. <`http://www.rfc-editor.org/info/rfc4120`>, July 2005.

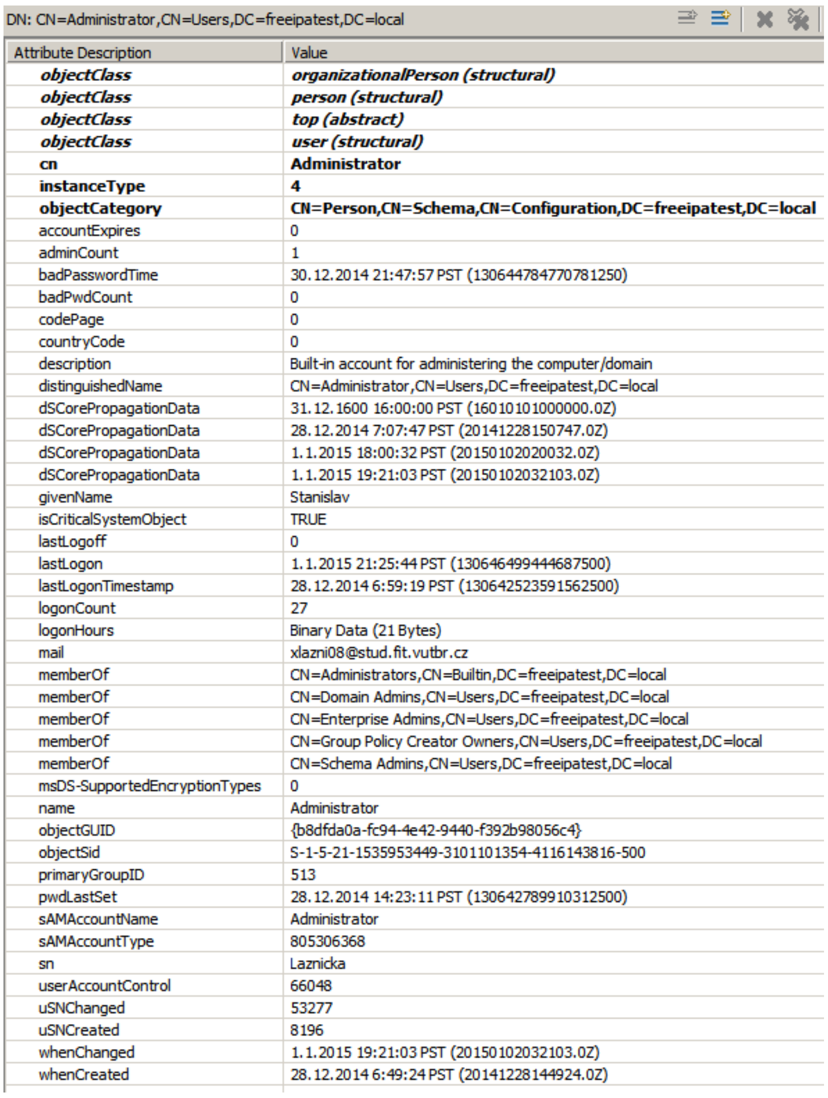
[17] SERMERSHEIM, Jim. *Lightweight Directory Access Protocol (LDAP): The Protocol*, RFC 4511. <`https://tools.ietf.org/html/rfc4511`>, 2006.

[18] SSSD. *SSSD source code* [software].[cit. 2015-05-17]. <`https://git.fedorahosted.org/cgit/sssd.git`>.

Appendix A

Pictures

Figure A.1 shows an example of a user account object in Active Directory.

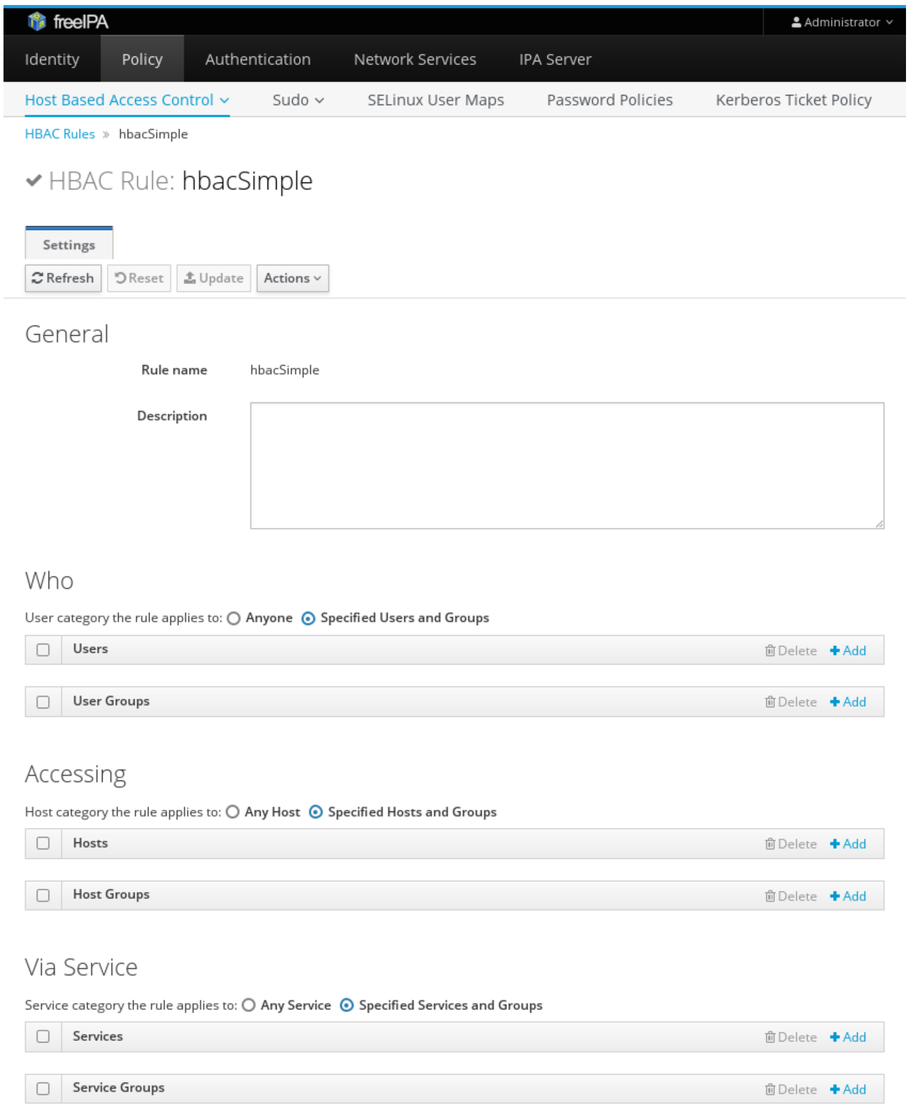


DN: CN=Administrator,CN=Users,DC=freeipatest,DC=local

Attribute Description	Value
objectClass	***organizationalPerson (structural)***
objectClass	***person (structural)***
objectClass	***top (abstract)***
objectClass	***user (structural)***
cn	**Administrator**
instanceType	**4**
objectCategory	**CN=Person,CN=Schema,CN=Configuration,DC=freeipatest,DC=local**
accountExpires	0
adminCount	1
badPasswordTime	30.12.2014 21:47:57 PST (130644784770781250)
badPwdCount	0
codePage	0
countryCode	0
description	Built-in account for administering the computer/domain
distinguishedName	CN=Administrator,CN=Users,DC=freeipatest,DC=local
dSCorePropagationData	31.12.1600 16:00:00 PST (16010101000000.0Z)
dSCorePropagationData	28.12.2014 7:07:47 PST (20141228150747.0Z)
dSCorePropagationData	1.1.2015 18:00:32 PST (20150102020032.0Z)
dSCorePropagationData	1.1.2015 19:21:03 PST (20150102032103.0Z)
givenName	Stanislav
isCriticalSystemObject	TRUE
lastLogoff	0
lastLogon	1.1.2015 21:25:44 PST (130646499444687500)
lastLogonTimestamp	28.12.2014 6:59:19 PST (130642523591562500)
logonCount	27
logonHours	Binary Data (21 Bytes)
mail	xlazni08@stud.fit.vutbr.cz
memberOf	CN=Administrators,CN=Builtin,DC=freeipatest,DC=local
memberOf	CN=Domain Admins,CN=Users,DC=freeipatest,DC=local
memberOf	CN=Enterprise Admins,CN=Users,DC=freeipatest,DC=local
memberOf	CN=Group Policy Creator Owners,CN=Users,DC=freeipatest,DC=local
memberOf	CN=Schema Admins,CN=Users,DC=freeipatest,DC=local
msDS-SupportedEncryptionTypes	0
name	Administrator
objectGUID	{b8dfda0a-fc94-4e42-9440-f392b98056c4}
objectSid	S-1-5-21-1535953449-3101101354-4116143816-500
primaryGroupID	513
pwdLastSet	28.12.2014 14:23:11 PST (130642789910312500)
sAMAccountName	Administrator
sAMAccountType	805306368
sn	Laznicka
userAccountControl	66048
uSNChanged	53277
uSNCreated	8196
whenChanged	1.1.2015 19:21:03 PST (20150102032103.0Z)
whenCreated	28.12.2014 6:49:24 PST (20141228144924.0Z)

Figure A.1: Administrator user object in Active Directory[12]

In the figure A.2, there's the page that shows when trying to add a HBAC rule in FreeIPA WebUI.



HBAC Rule: hbacSimple

General

Rule name: hbacSimple

Description:

Who

User category the rule applies to: Anyone [✓ Specified Users and Groups]

Users

User Groups

Accessing

Host category the rule applies to: Any Host [✓ Specified Hosts and Groups]

Hosts

Host Groups

Via Service

Service category the rule applies to: Any Service [✓ Specified Services and Groups]

Services

Service Groups

Figure A.2: FreeIPA - Add a HBAC rule[7]