BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FREEIPA – URI BASED ACCESS MANAGEMENT

FREEIPA – SPRÁVA PŘÍSTUPU DLE URI

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. LUKÁŠ HELLEBRANDT

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. TOMÁŠ KAŠPÁREK

BRNO 2016

Master Thesis Specification/18243/2015/xhelle04

Brno University of Technology - Faculty of Information Technology

Computer Centre Academic year 2015/2016

Master Thesis Specification

For: **Hellebrandt Lukáš, Bc.**
Branch of study: Information Technology Security
Title: **FreeIPA - URI Based Access Management**
Category: Operating Systems

Instructions for project work:

1. Study the central identity management, open-source identity and authentication provider FreeIPA and its host-based access control (HBAC) feature, focusing on its use for authorization for Web applications.
2. Design an extended access control mechanism which will take the URI being accessed by the user into consideration.
3. Implement this extended access control mechanism in FreeIPA and other related components.
4. Demonstrate the solution on some existing Web application and discuss gained improvements.

Basic references:

- FreeIPA, http://freeipa.org/

Requirements for the semestral defense:
Items 1 and 2.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Kašpárek Tomáš, Ing.**, CC FIT BUT
Beginning of work: November 1, 2015
Date of delivery: May 25, 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

Dušan Kolář
Associate Professor and Head of Department

Abstract

The goal of this thesis is designing and implementing access management based on URI of the requested resource. Host Based Access Control in the identity management tool FreeIPA was used as a basis for implementation. Furthermore, it was necessary to enhance the related infrastructure, namely the SSSD tool. The authorization module for Apache HTTP Server was used as an example of the application using URI-based HBAC. The main solved problem was design of the infrastructure for communication of the necessary parameters and strategy proposal for evaluating HBAC rules which define the access rights. The complete solution was demonstrated on the example of securing an instance of the web application Wordpress.

Abstrakt

Cílem práce je navržení a implementace řízení přístupu na základě URI požadovaného zdroje. Pro implementaci bylo jako základ použito rozšíření Host Based Access Control v nástroji pro správu identit FreeIPA. Zároveň bylo třeba rozšířit související infrastrukturu, především program SSSD. Jako příklad aplikace využívající HBAC na základě URI byl implementován autorizační modul pro Apache HTTP Server. Zásadním řešeným problémem byl návrh infrastruktury pro komunikaci nezbytných parametrů a návrh strategie vyhodnocení HBAC pravidel definujících přístupová práva. Kompletní řešení bylo předvedeno na příkladu zabezpečení instance webové aplikace Wordpress.

Keywords

FreeIPA, URI, SSSD, PAM, LDAP, authorization, access control, Host based access control

Klíčová slova

FreeIPA, URI, SSSD, PAM, LDAP, autorizace, řízení přístupu, Host based access control

Reference

HELLEBRANDT, Lukáš. *FreeIPA – URI based access management.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Kašpárek Tomáš.

FreeIPA – URI based access management

Declaration

I hereby declare that I am the sole author of this master's thesis. I have written it under supervision of Ing. Tomáš Kašpárek. I have cited all the sources I have used.

. . . .
Lukáš Hellebrandt
May 20 2016

Acknowledgements

I would like to thank Ing. Tomáš Kašpárek for supervising my work.

Contents

List of Figures

Chapter 1

Introduction

This thesis is about extending functionality of FreeIPA [10] - an identity, policy and audit management tool. Our goal will be adding support to manage access to resources in services provided by FreeIPA clients, based on the resources' URI. We will begin by describing our motivation and goals before we can proceed to technical details and a concept.

The thesis and related work is being worked on in cooperation with Red Hat, Inc. [28].

1.1 Motivation

Before describing our exact requirements on the newly developed functionality, we will discuss our motivation. It highly affects the expectations on the new system and, together with already existing solutions of subproblems, is the main contribution to the final specification.

Reasons to engage in extending FreeIPA with URI-based access management can be divided into two main categories:

1.1.1 General need for URI-based access management

General need for this tool stems from many existing services and their complexity. There may be multiple different services running on a single host, just as there may be multiple instances of the same service. Different services have different configuration format and configuration tools. They also have different level of security and different (if any) options of solving the access management on application level.

Furthermore, security should always be ensured on multiple levels. Adding service-level authorization can add another layer of security. For example on a webserver, it might be beneficial to check user permissions *both* on application level (the web application checks user permissions which has to be ensured explicitly by its programmer, often based on local application data rather than centrally managed data) and on service level (HTTP server checking permissions on its own, regardless of application).

Due to this complexity, possible complications and high risk of them going unnoticed, we identify general need to manage access rights accross different services. We need a solution that can be configured consistently on different services or different instances of a service. It also needs at least basic user friendliness and should be widely accepted and accessible – at least the last two demands can be appropriately supported by using open source [60] [50] solutions as a base for further work.

A typical use case can be the following: There is a service (for example a webserver) on some host system. This host system is also registered as a client of the FreeIPA identity

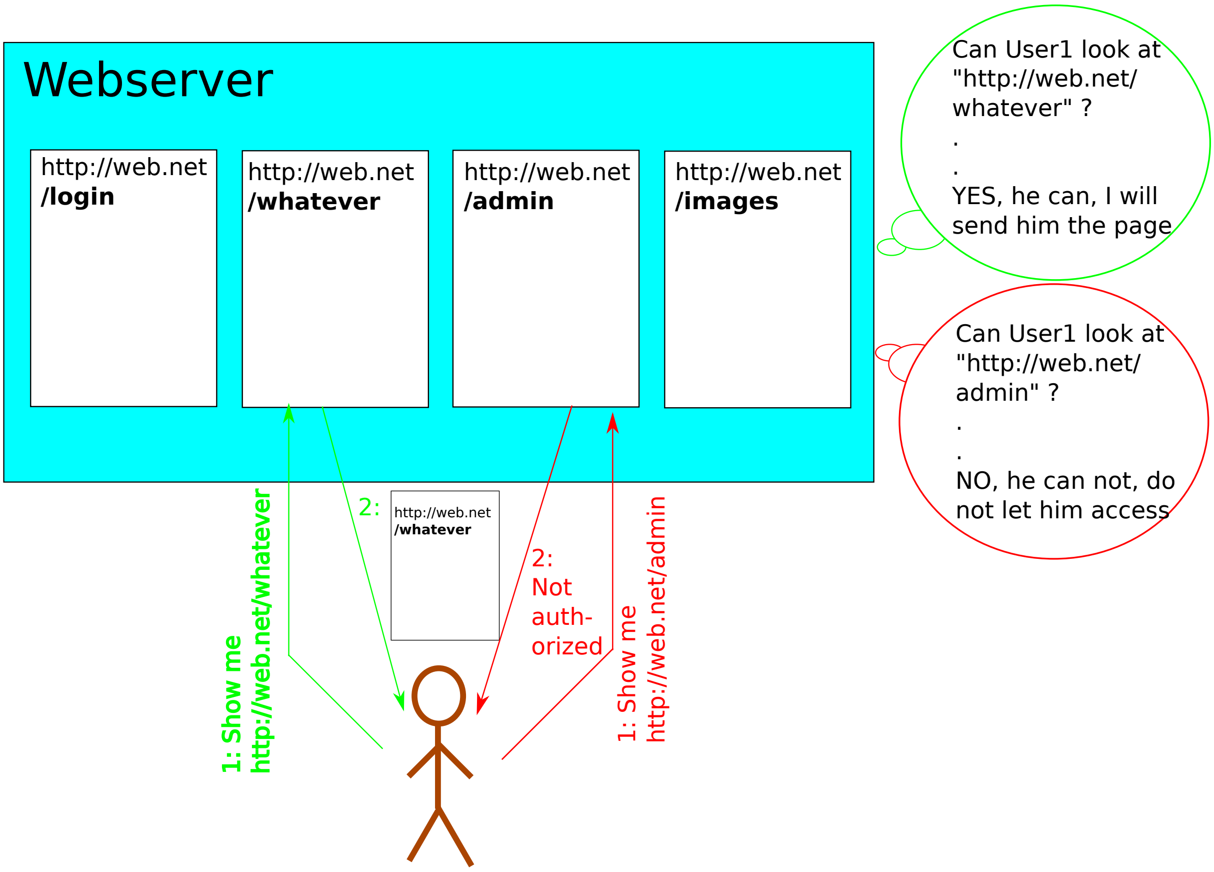

Webserver
http://web.net /login
http://web.net /whatever
http://web.net /admin
http://web.net /images
Can User1 look at "http://web.net/ whatever" ? . . YES, he can, I will send him the page
Can User1 look at "http://web.net/ admin" ? . . NO, he can not, do not let him access
1: Show me http://web.net/whatever
2:
http://web.net /whatever
2: Not auth- orized
1: Show me http://web.net/admin

Figure 1.1: Diagram showing a problem – authorization based on URI

management tool, which is (usually) running on another system. Some user on some other system wants to use this service (view a webpage). The server (webserver) needs to somehow verify whether the user is authorized to access this resource (part of the web). The resource is conventionally described by URI. Therefore, we need to determine whether a user can access some URI through some service on some host, as shown in figure 1.1. A typical approach would be a database of users and their permissions, usually only used by that application and being accessed on the (web) application level.

However, we want the authorization to happen against some centrally-managed database, like FreeIPA. We also do not want to handle authorization explicitly in each (web) application. In our use case, we want the (web) server application (often Apache HTTP Server [6]) to handle authorization against the FreeIPA server which the server system is registered to as a FreeIPA client.

To achieve this, we are missing a feature on both ends – FreeIPA does not currently store URI-related information in its authorization rules and Apache HTTP Server is not able to handle URI-aware authorization against FreeIPA. Users can reasonably expect there is a way to cover this use case and similar use cases in FreeIPA – authorization is one of its main goals. We have, therefore, a good reason to want to include this functionality in FreeIPA.

1.1.2 Specific requests and needs

Another reason to work on this problem – and for Red Hat to be involved – are needs of specific users. These needs are expressed, among others, by Red Hat's customers requests. Multiple companies demand that they are able to authorize access based on URI, as we can see in the ticket [29] on the FreeIPA Trac [37] ticketing system [11], which originates from a RFE [36] requested by a Red Hat customer. The customer is using a third-party authentication / authorization software and wants to replace it by FreeIPA. Lack of this specific functionality, however, keeps them from using FreeIPA which illustrates real demand for URI-based authorization.

There is demand for the URI-based authorization to work on basis of existing solutions, such as FreeIPA, on Red Hat's platform [55] [54]. If accepted by upstream, that also means the improvement will be made available for broad audience as FreeIPA is an open source project compilable for multiple architectures and platforms [30] [23].

1.2 General requirements

Based on above, we can put together this very general and informal specification of requirements we will try to fulfil:

- Support authorization of user accessing certain part of service based on URI – store necessary data, manage it in a user-friendly way, be able to decide based on it
- Accessible, accepted solution
- Usable on a typical Linux system used as a webserver

In the following chapters, we will describe existing technologies and protocols and formulate more exact specifications. Theoretic analysis and description of technologies (chapters 1 through 6) is mostly taken from the semestral project. Based on it, we will elaborate on possible approaches, thoroughly describe the concept and ultimately show our implementation.

Chapter 2

Identity and access management

With rapidly growing usage of electronic services, such as information systems, cloud systems, and other means of accessing remote resources, together with growth of user base of such services and companies using them, a need has arisen to manage these users' identities and access rights externally. There are large-scale databases of users for enterprise-level companies. There are also many services (or instances of service) used, sometimes on one host, sometimes on many.

In such a large scale, it would be extremely inefficient and even insecure to store user information (including data necessary to authenticate the users and for non-trivial applications to authorize their access to some resource) locally for each service (or each host) that needs it. It would require too much time and effort to maintain this data and keep it up to date. It would also be easier for an unauthorized user to secretly alter it, perhaps with malicious intent. Just mere unauthotrized access (without modifying anything) to this data can pose a security risk, too. Unnecessarily consumed resources are one more reason why this could be considered inefficient.

Rather than storing user data on multiple places without any central management, organizations choose to use *centralized identity and access management* applications. Purpose of those is to centrally store data necessary for authentication of users and, using network connection to hosts of resources, provide authentication services. While there is a system for remote authentication of users, it is reasonable to require another functionality closely related to authentication – these systems can also store data necessary for authorization of users. This data can sometimes be exposed to anyone, sometimes the only application that has access to it is the identity management system itself, which then autonomously decides to authorize or not authorize a user to certain resource and merely answer the result. Identity and access management systems may also store some additional data so they can provide more information about user, which can be used for example for automated registration of the new user based on this data – the user does not need to fill them manually. These identity management applications are designed for easy maintenance (usually with some GUI). Being centralized also makes updating the data easier and increases security – data kept in just one place is easier to be kept safe. It is also easier to audit such a system and verify its data is in expected state.

2.1 Roles

To understand the benefits of an identity management system, it is important to understand three basic roles an entity can appear in with regards to that system. These are the main entity types used in an abstract, simplified schema of an identity management system:

- *User* – information about many users is stored in the databse of the identity management server and is a basic entity in the schema. The whole system is here for maintaining and sharing some information about users. User has some identity and the means of proving it. User also has access rights assigned to certain resources.

  Users can be divided into *groups*. Groups can have some attributes of a user, like access rights. Every user in this group then posesses these attributes. In some configurations like trust or federalized setups, groups may be the only entities avaialable to the access management system.

- *Service* – the resource. Usually runs on some host and asks the identity management server to authenticate the user. Also usually asks the server to authorize the authenticated user, although in some cases, mere authentication can be considered sufficient to be authorized to access, based on the application. We will later use the word “principal” for users together with services.

  Particular service can be identified by multiple means (tuples), where *Host* is the host providing the service, *Port* is the number identification of a port the service is running on, *PAM service* is an identifier of a service by PAM standard, and *URI* is a resource identifier:

  - (Host, Port)
  - (Host, Port, PAM service)
  - (Host, Port, PAM service, URI) – can identify particular part of service, like “Webserver on system `www.host.net` with the URL of `http://www.host.net/login/page.php`”.

- *Administrator* – the person responsible for the identity management system. He is the one who benefits from centralization because he does not need to update the data on every host with some resource requiring authentication or authorization. Also, the system is less prone to administrator mistakes. Administrator’s job is keeping the user information and access rights up to date.

2.2 Example of use

Imagine there is a large organization with a lot of users. The organization uses many computers, some of them serving as a host of some services. Some of these services (or parts of services) are only to be accessible by certain users. Permissions of users can vary accross hosts and services. The following examples are what a simple use case scenario would mean if the organization stored the user data locally on each host (depicted in figure 2.1), compared to the organization using central identity management service (depicted in figure 2.2):

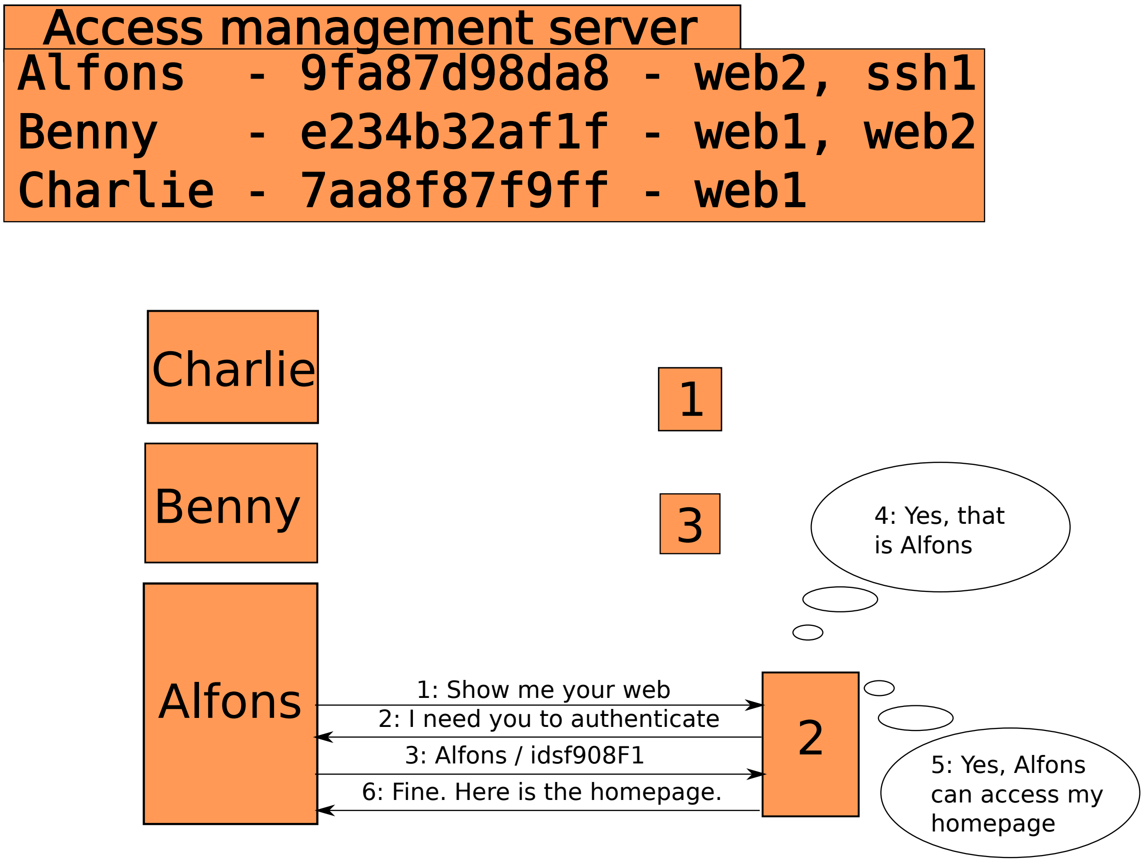

Access management server
Alfons - 9fa87d98da8 - web2, ssh1
Benny - e234b32af1f - web1, web2
Charlie - 7aa8f87f9ff - web1
Charlie
Benny
Alfons
1
3
2
4: Yes, that is Alfons
1: Show me your web
2: I need you to authenticate
3: Alfons / idsf908F1
6: Fine. Here is the homepage.
5: Yes, Alfons can access my homepage

Figure 2.1: Conceptual model of an user connecting a webserver *not* using identity management system for authentication and authorization. This web application uses its own database and user authenticates directly with it. 1, 2, 3 are hosts; Alfons, Benny and Charlie are users.

2.2.1 Locally stored access rights

- When user `U` (his service client) connects host `H` requesting access to service `S`:

  1. `H` looks into its internal database and searches for the user. If it can find him, the host `H` verifies the user's identity based on some attribute of the user (Which means H has access to this attribute – not only may it be stored in H's database, U provides it to H on every authentication. Even if, for example, a password is not stored in plaintext, the user must be careful to not use the the same password on multiple services as they receive it every time the user authenticates).
  2. If the user is authenticated, the host looks up its internal database to verify the user is authorized to access the service. These two things can also happen on service level but the important part is that the database is stored on the host `H`.
  3. If `U` is authorized by `H`, `U` accesses `S` on `H`.

The communication is depicted in figure 2.1. No communication with any central management server is required. All the communication, including authentication, is carried out directly between `U`'s client and `H`.

- The above example is as simple as it can get. Consider, however, another example – when an administrator A needs to change user U's password. In that case, A needs to modify databases on every single host, for every single service (if they do not share the database), changing U's password in that database.

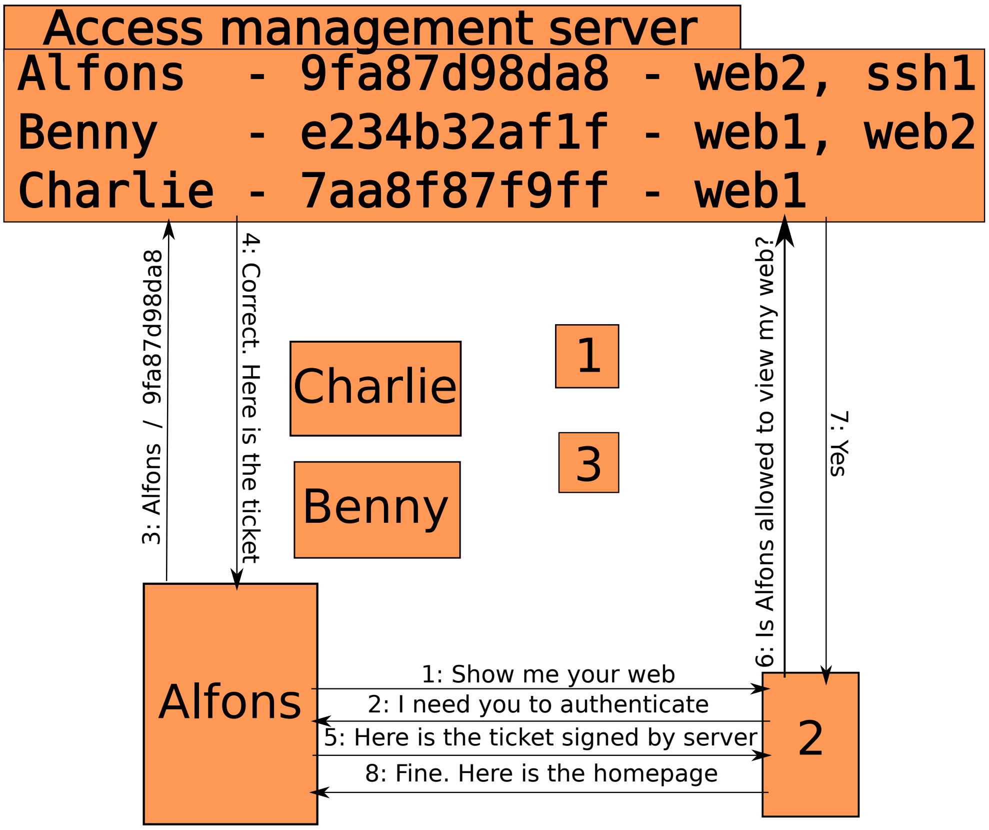
```
forall Hosts do
    forall Services on Host do
        update password
    end
end
```

2.2.2 Central identity management

- When user U (his service client) connects host H requesting access to service S:

  1. U connects H
  2. Service S on host H authenticates U based on some information on the identity management server. No user information (not even a password hash) is stored on H, the identity management server merely informs H that a user has been authenticated and provides H U's identity. This conceptual model is usually achieved via a *ticket* – a proof that a user has authenticated against the identity management server, signed by that server. It means that the server does not provide the information directly, it is a job for the user's client instead to authenticate against the server and provide this ticket. We will describe this process more thoroughly in the section about Kerberos.
  3. If S requires specific user rights for U to access, H (usually on service S's level) asks the identity management server for authorization of U to the given resource S. Again, this information is not stored locally, so the identity management server does one of the two things:
     - It either merely responds "authorized"/"unauthorized". No data are let out whatsoever and there is minimal overhead on H's side. However, H is in no way aware of what the identity management server's response is based on and needs to always ask it; it can not do *any* decision locally.
     - Or the identity management server sends non-sensitive data H needs to decide itself. This can be practically achieved (and it is the case of our work) by direct communication between the identity management server and the host providing the service, in contrast with authentication where a user's client and a ticket are involved. This can have performance implications as the data necessary for authorization are let out, thus they can be cached by S and in some cases, S does not need to always ask the identity management server for authorization.
  4. U, if authorized, gets required resource from S.

An example of communication is depicted in figure 2.2. This scheme is somewhat more complicated than in the first case, it involves one more side – identity management server. It has security advantage, though: no data is stored locally, not even password hashes, they are just in one place. That place is easier to secure and only lets out the information that accessing hosts necessarily need to know: user's identity and whether he is authorized to access some resource.



```
Alfons  - 9fa87d98da8 - web2, ssh1
Benny   - e234b32af1f - web1, web2
Charlie - 7aa8f87f9ff - web1
```

3: Alfons / 9fa87d98da8
4: Correct. Here is the ticket
Charlie
Benny
1
3
6: Is Alfons allowed to view my web?
7: Yes
Alfons
2
1: Show me your web
2: I need you to authenticate
5: Here is the ticket signed by server
8: Fine. Here is the homepage

Figure 2.2: Conceptual model of an user connecting a webserver using identity management system for authentication and authorization

- When administrator wants to change a password, he simply changes it in the only location it is stored in: *on the identity management server*. From that time on, the password is updated and every host is affected – the hosts do not verify the password locally, they just receive information about authentication or authorization from the identity management server that is up to date.

In conclusion, we can say that identity management systems improve security, save maintenance effort and while setup might be harder for a single host, they save a lot of time and even disk space when used in enterprise-level environments. They achieve this by centralizing information about users and their permissions. They are designed never to disclose sensitive information and are an important part of enterprise-level companies' security measures.

Chapter 3

Technologies and terms used

Here, we will describe some terms and technologies necessary to understand the rest of the work.

3.1 URI

The "URI" abbreviation stands for "Uniform Resource Identifier". It is specified by the IETF [16] RFC 3986 [43] from year 2005 which updates an older RFC 1738 [41] specifying URL ("Uniform Resource *Locator*").

It is a compact set of characters that identify an abstract or physical resource. It can be either an URL ("Uniform Resource Locator", as in the updated RFC 1738), URN ("Uniform Resource Name") or both. [43]

- *URL* is a way of providing means of locating some resource by its primary access mechanism. [43] An example could be an URL of some web page – the usage mostly known even to unskilled users. URL "`http://www.fit.vutbr.cz`" provides a way of locating the FIT VUTBR web page rather than describing what it is or naming it.

- *URN* is used to provide name of some resource. It is URI that is required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable, and any other URI with the properties of a name. [43] Its syntax is specified by RFC 2141 [52]. For example an ISBN number of a book is an URN – a unique identification of a book. It is not an URL because it does not in any way show us how to locate the book.

Therefore, we can say that URI is a way of identifying some resource either by its location (URL) or its name (URN) or both.

In our work, we use URI as an attribute for deciding whether particular authenticated user is authorized do access the resource identified by the given URI. We do not need to differentiate between URL and URN. In most of the services, however, it is customary to use URL. This has a practical reason – it tells us exactly where and how to find a resource.

Our mostly used example, a web server, identifies its resources by URLs (the "address"). This is also the case for which we will – as described later – develop a service-side module. The enhancement is, however, in no way limited only to web servers or URLs and is more general.

3.1.1 URI parts

The RFC specifies a URL using grammar in Augmented Backus-Naur Form [45]. We will show the basic parts of URI in a more informal and incomplete way.

Generally, URI can consist of these main parts:

- *Scheme* – a specification for assigning identifiers to be used. In case of webserver, it is usually either `http` or `https`.
- *Hier-part* – which consists of:
  - *Authority* – the authority that governs the name space in the rest of URI, delegated to that authority. If URI contains some authority, the hier-part of URI must start with "//", otherwise it must not start with "//".
    - *userinfo@* – username and scheme-specific information about how to gain authorization to access to the resource. It can be omitted.
    - *host* – the identification of a system – IP literal ("IPv6 address"), IPv4 address or a hostname.
    - *:port* – decimal number identifying a port the service is running on. Can be omitted and the scheme can specify default port.
  - *Path* – data – usually hierarchical – identifying the resource withing the scope of URI scheme and a naming authority, if it exists.
- *Query* – non-hierarchical data identifying the resource within the scope of URI scheme and a naming authority, if it exists.
- *Fragment* – identification of certain part of a resource.

[43] These parts form URI in the following way:
`URI = scheme : hier-part [ ? query ] [ # fragment ]` As examples, we show one URN and one URL and how they are divided into these mentioned parts:

- Example of an *URN* is `urn:isbn:0-330-25864-8`, identifying a book by its ISBN. The part before the first colon is *scheme*, saying that the following is an URN. The rest is a *path* part, identifying that book.
- In our case, however, we are mostly going to work with *URLs*. These are used to identify a web server and certain web page on that server. An example URL is `https://www.youtube.com/watch?v=oHg5SJYRHA0`.
  - `https` is *scheme*. It can also be *http* for a webserver.
  - `:` delimits scheme from the rest.
  - `//` must be in URI because it contains *authority*.
  - `www.youtube.com` is *authority* – a web server identification using hostname.
  - `/watch` is a *path*, it must start with `/` and specifies the part of web we require.
  - `?v=oHg5SJYRHA0` is a *query* – the data that further identifies the resource in a non-hierarchical manner. In this case, the query specifies a video the user wants to view.

This URL is missing some parts that are not mandatory, but can be specified. We used a default port, so we did not have to specify it. A default port is specified by the `https` *scheme* and is `443`. We also did not specify fragment. We wanted to specify a web page as a whole, not some part of it.

3.2 Authentication, authorization

Authentication and authorization are the terms used in a general "AAA" security model. AAA stands for Authentication, Authorization and Accounting. It is an abstract model of verifying who the user is, what resource he can access and what he did with that resource [59]. We are, in this work, interested mainly in the second A, supposing the first A has already been completed at point the service starts to act on authorization. More formally, authentication and authorization are:

- *Authentication* is a process of verifying a user's identity. Authentication is usually based on:
  - What we know (e.g., a password)
  - What we have (e.g., a token)
  - What we are (e.g., a fingerprint)

  For example, showing your ID to the gate keeper so he can verify that you actually are who you claim to be is authentication.
- *Authorization* is a process of verifying whether a user is allowed to access some resource. An example could be the gate keeper from the previous example looking through a list of people allowed to enter the building. As you can see, to *authorize* an user to access some resource, the authorization authority also needs to *authenticate* the user in some way or have information about user's identity from trusted third party.

In our work, we do not solve anything *authentication*-related as we expect the user is authenticated already. There are multiple ways to achieve this, particularly for the Apache HTTP Server we can mention Kerberos authentication and user/password authentication. Given the authenticated user's identity, we will have to decide whether he is allowed to access some resource identified by an URI.

3.3 Kerberos

Kerberos is not a key feature for this work. We have, however, mentioned it multiple times, it is used in FreeIPA and we are going to use it as the authentication step we will expect to be done already before authorization. Also, we might later decide that Kerberos will carry some of the authorization data, albeit it is not capable of handling authorization on its own.

Kerberos started as MIT's Project Athena [44]. The current version of Kerberos is Version 5 [49], development of which began in 1989. It is a distributed authentication service that allows a process of a principal (user) to prove its identity to a server [53]. It prevents authentication data from being discovered by any party except for user and Kerberos server. It optionally provides integrity and confidentality of communication between client and server.

3.3.1 Basic Kerberos protocol

In this simplified version of a basic Kerberos protocol, we will show how a user can authenticate in cooperation with a Kerberos server. We will also learn how a user proves he has been authenticated to the server he requests a resource from and that a server can prove its own identity back to user.

Parties:

- *Principal* is a party whose identity is being verified [53].
- *Verifier* is a party who demands to authenticate principal.
- *Authentication service* is a party who authenticates principal for verifier.

The basic protocol consists of two parts. The first one is authenticating principal with authentication service, the second one is using the fact principal has authenticated to actually prove the principal's identity to the verifier.

- *Authentication request and response* – the phase where a principal communicates with the Kerberos authentication service. It proves its identity using its password. The authentication service, upon successfull authentication, responds with a *ticket*

  A *Kerberos ticket* certificate from the authentication server contains, among other information, a random *session key*, a checksum and the name of the principal to whom the ticket has been issued [53]. It can also contain some additional information about the principal. It is encrypted using a key shared only between the *verifier* requesting the authentication and the Kerberos authentication service. It is a property of the used encryption algorithm [53] that when a message has been changed or a key is different than the key that has been used for encryption, the resulting decrypted data will not make sense and the checksum will not fit. A verifier can thus believe this ticket even if it does not receive it directly from the Kerberos authentication service. In fact, the ticket is merely responded back to the principal (which can *not* modify it in any way as the encryption key used is *only* shared between the authentication service and the *verifier*) and the principal can further send it to the verifier to prove its identity.

  In figure 3.1, in phase 1, the principal sends an authentication request to the Kerberos authentication service. In phase 2, the principal receives the authentication response, including its ticket and a session key.

- *Application request (and response)* – when the principal receives its ticket, it uses it to prove its identity to the verifier. The ticket includes the principal name and a session key. Additionally, there is also current time, checksum and some other data sent. Thus, verifier knows the identity and can communicate with the principal in a safe manner. Due to cryptographical properties of the data included in ticket and authenticator, the verifier can actually trust the ticket data and consider the principal authenticated. There can be an optional application response that is used in cases where a verifier also needs to authenticate to the principal.

  After these steps (optionally omitting the response), the two parties can further communicate safely as their identities have been verified. The authentication request is shown in figure 3.1 in step 3, the authentication response is in step 4.

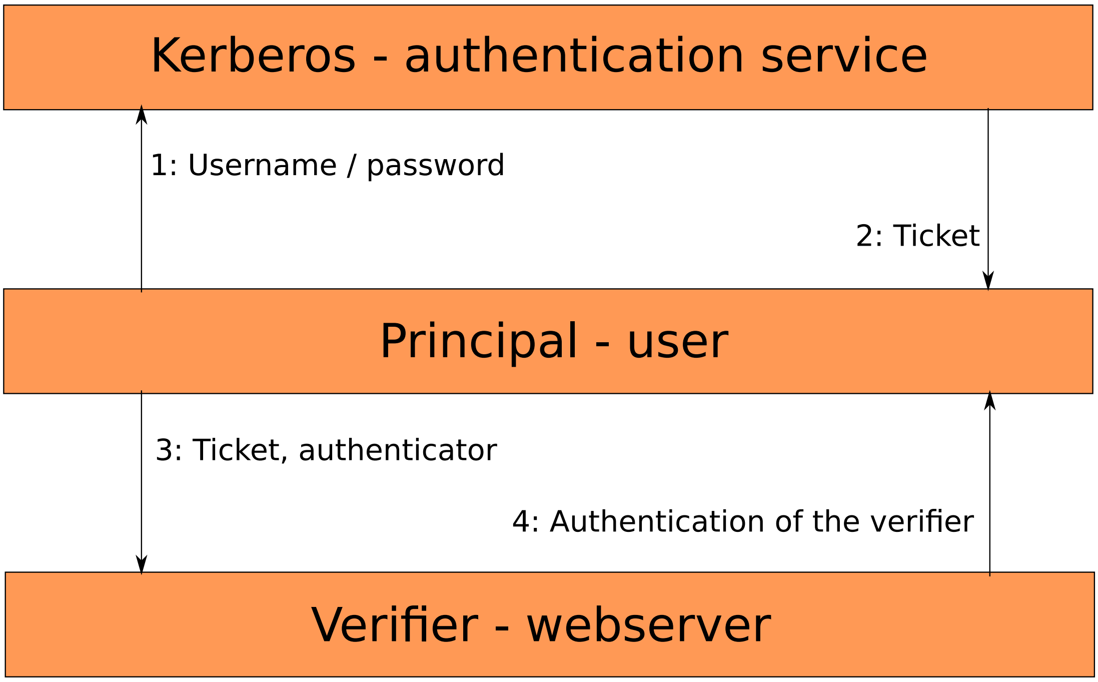


Figure 3.1: Simplified model of basic Kerberos authentication request/response and application request/response

3.3.2 Ticket granting service

Mainly for user convenience, the scheme above is not actually how Kerberos works nowadays. Instead of user providing a password each time he needs to access some service to get a ticket, user provides another ticket – *a ticket granting ticket* – which is the only ticket initially received from the Kerberos server using user/password authentication.

The user receives a ticket granting ticket (TGT) when he first needs a service ticket, after explicitly asking for it or after logging in, depending on his system. He contacts a Kerberos Authentication Service and gets the TGT based on user and password provided. This ticket can then be used to obtain another ticket, a *service ticket*, from Kerberos Ticket Granting Service, whenever the user needs to access some specific service. He then receives a ticket as shown in figure 3.2, which means he uses the TGT instead of a password in figure 3.1 in the previous section. This is possible until the TGT expires, after that it must be renewed.

3.3.3 Additional principal data and Kerberos

A Kerberos ticket contains, among other fields, the optional field *Authorization-Data* [12]. Different implementations of Kerberos use this field differently. The Kerberos service itself does not try to interpret it; interpretation is left up to the service requesting credentials [12].

We could use this Authorization-Data field as a carrier of data that is important to us, because – as we will learn – Kerberos is used in FreeIPA. After all, the field is *named* as to serve the purpose of containing authorization data. That would mean the service does not even need to be aware of FreeIPA – mere Kerberos-awareness would be enough. Changing the field's content may, however, be a big problem for implementations using this optional field and this change might impact a lot of applications. There are also some

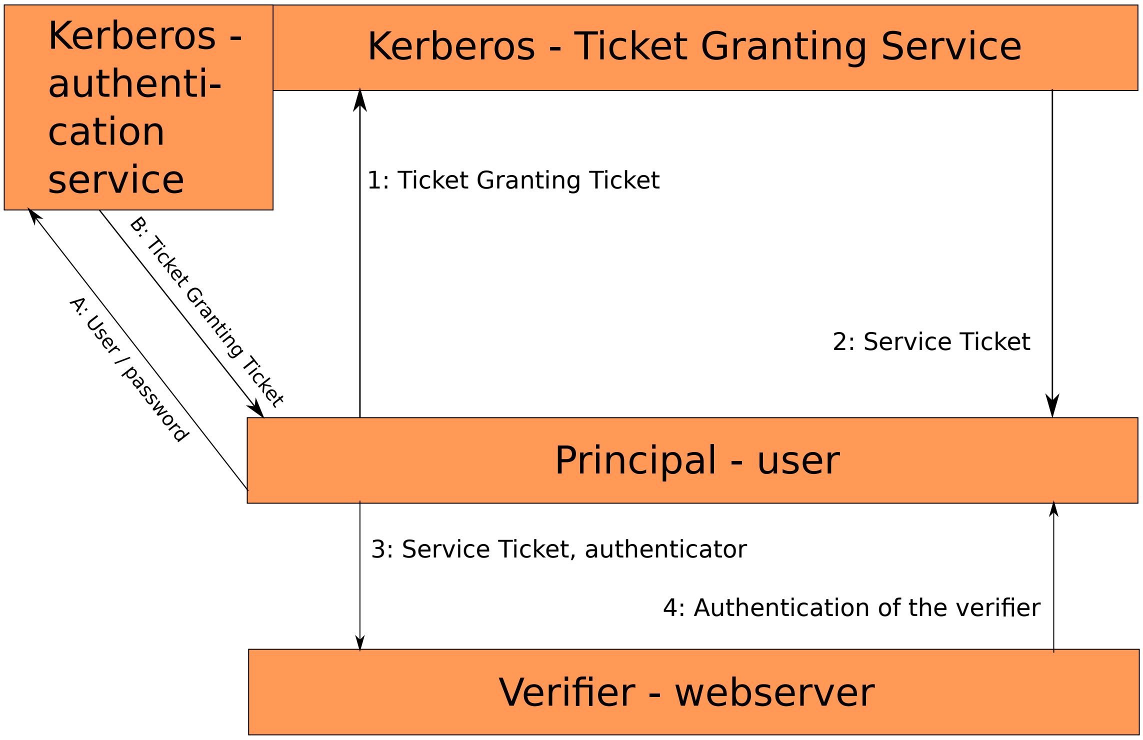

Kerberos - authentication service
Kerberos - Ticket Granting Service
1: Ticket Granting Ticket
B: Ticket Granting Ticket
A: User / password
2: Service Ticket
Principal - user
3: Service Ticket, authenticator
4: Authentication of the verifier
Verifier - webserver

Figure 3.2: Simplified model of basic Kerberos authentication request/response and application request/response using Ticket Granting Ticket

FreeIPA-related problems with this approach, as we will discuss in further parts of this text.

3.4 LDAP

LDAP is a directory service. Directory service is a central repository for storing and managing information [17]. It is often hierarchical. Directory service is similar to database, but typically contains more descriptive, attribute-based data – the data that are more often read than written [39].

The data in LDAP is hierarchically organized and has its object type. LDAP provides a way to read, write, modify and search data in the directory. The object type is similar to "data types" in databases. They define the interpretation (and very often semantics) of attribute value. The object type is identified unambiguously by *OID* (Object Identifier). The OID's are defined as hierarchically structured integers, defined in *X.690 standard* [48].

3.4.1 Adding custom data

To add an attribute type, we use the `attributetype` directive. We can specify description, matching, syntax etc.. Matching specifies how different entries of this attribute should be interpreted in certain situations, for example how they should be tested for equality (`EQUALITY caseIgnoreMatch`, for example). The `SYNTAX` specifies how the value should be represented. For example, the syntax with OID of `1.3.6.1.4.1.1466.115.121.1.15` denotes a directoryString (UTF-8 string) [31].

To specify a new object class (and possibly use the newly specified attribute in that class), we use the `objectclass` directive. It specifies its position in hierarchy (`SUP`, `DESC`), attributes, whether they are mandatory (`MAY`, `MUST`) etc.

To save and exchange the data and schema between LDAP servers, we can use *LDIF files* [1].

3.5 FreeIPA

FreeIPA is an integrated security information management solution combining Linux, 389 Directory Server, MIT Kerberos, NTP, DNS, Dogtag (Certificate System). Its interface consists of a web interface and command-line administration tools. [2]

The IPA abbreviation stands for *Identity, Policy, Audit*:

- *Identity* in the name means it is an implementation of the Identity management server concept described in chapter 2.
- *Policy* basically means authorization of access to some service based on some information about the service itself, the client etc. This will be described later. A key term for the "policy" part are sudo rules, Host Based Access Control (HBAC) rules and authorization based on these.
- *Audit* essentially means logging and viewing the history of actions of entities. This component is deferred and unrelated to our work.

FreeIPA's goals are providing an identity management solution on enterprise level, with user-friendly web interface, consisting of open-source projects. It stores data related to user authentication, authorization, HBAC rules, and others.

3.5.1 Architecture

High-level architecture of FreeIPA is shown in figure 3.3. FreeIPA is a set of multiple applications used together to make an easy-to-use, robust solution for identity and policy management. It consists of the following parts:

- *389 LDAP directory server* serves as a backend. It contains information about users, machines, domain configuration, policy rules etc. Various *ipalib plugins* are used to expose and modify the data in the directory, either by means of console commands or through WebUI.
- *Kerberos KDC* is used for authentication handling in FreeIPA. It uses the data in LDAP directory and is a classical Kerberos, just configured for FreeIPA. It is managed through FreeIPA means, Kerberos tools are unaware of FreeIPA! [22].
- *Apache HTTP server* is used for WebUI and API. While it is possible to maintain FreeIPA solely by command line tools, a WebUI is more intuitive. The WebUI is modular and massively utilizes Javascript and XML-RPC API to manage LDAP directory's content.
- *DNS* serves as a typical DNS for the domain. SSSD clients are configured to use it for service discovery [22]. DNS is not interesting for our purposes.

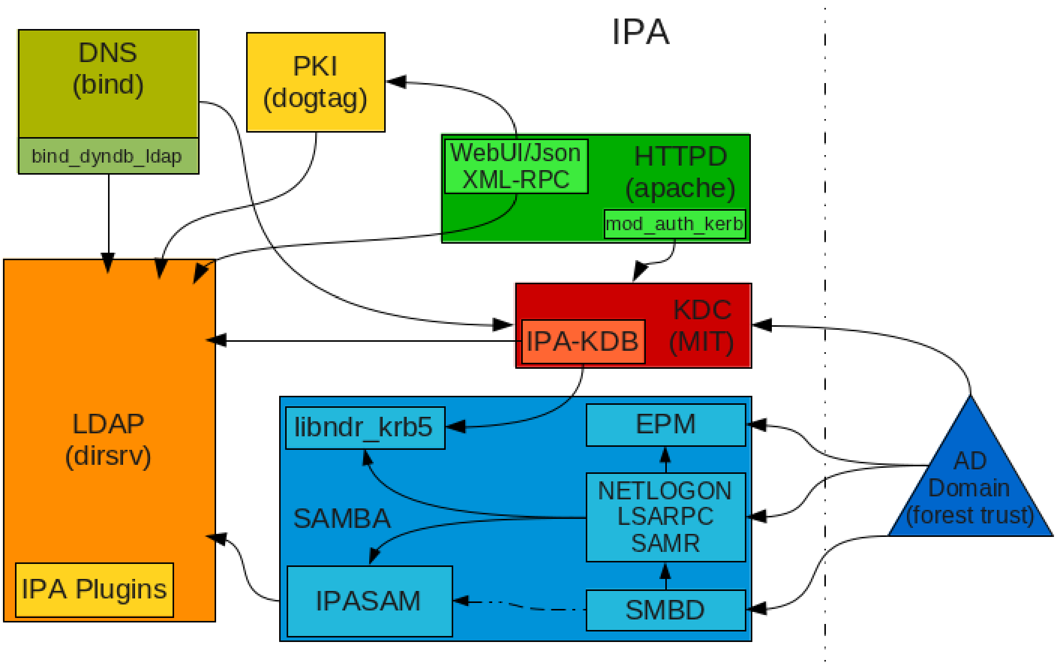


Figure 3.3: FreeIPA architecture [18]

- *NTP* is used to synchronize time within domain, which is required by some services – for example, Kerberos uses time stamps in tickets. NTP is not interesting for our purposes.
- *Samba* implements MS-RPC services and is not interesting for our purposes.
- *Dogtag Certificate System* is a PKI service. It includes a certificate authority [14] issuing certificates to services and CRL and OCSP services. It is not interesting for our purposes.

FreeIPA defines a domain of controlling servers and client machines. All its components work together as a compact tool with unified interface (either console commands or WebUI). As these components are aware of each other, it allows for more consistency, less administrative overhead and predictable environment through servers and enrolled clients, which are part of the domain.

It is also still possible to access each part of this infrastructure separately - for example a service can only use Kerberos or you can access LDAP data directly.

3.5.2 Authentication, Kerberos in FreeIPA

The *I* part is ensured mainly by Kerberos. Kerberos uses data from LDAP for authentication. The LDAP itself defines and enforces access controls for the Kerberos data stored within it [14].

The services which only need authentication do not need to be aware of FreeIPA at all and they can use Kerberos transparently. They can get additional (e.g., authorization) data directly from LDAP, for example HBAC rules.

3.5.3 Host Based Access Control

The *P* part – policy, which can also be understood as authorization – is represented by a feature called Host based access control (HBAC). This is the key functionality of FreeIPA for our purposes and is most likely to be modified.

Host Based Access Control is a way of defining and enforcing permissions of users. The key entity in HBAC is a *HBAC rule*, which specifies which user can access which service on which machine. On authorization request, these rules are evaluated one by one and if any of them matches, the action is authorized [3].

There used to be *deny rules*, too. They have been dropped, however. Reasoning behind this is that in every sane environment, all access rules should be defined by whitelist [27]. This might be a complication in case of adding URI to the rule. A typical example could be a need to allow access to the whole web *except* the administration part which contains "`/admin/`" somewhere in the path. In some cases, that can be worked around by whitelisting every other possible path. This might benefit from interpreting an URI in the rule as prefix, not the whole URI. In other cases, number of these workaround rules might be infinite, even if we interpret the specified URI as a prefix. This might be a major limitation or might become a cause to re-evaluate the reasoning behind dropping deny-rules.

HBAC rules

HBAC rules themselves are very simple entries. They specify access permissions based on 3-tuple *(User,Host,Service)*. When all of these match, the rule as a whole is matched and access is allowed.

To take URI into consideration, we will need to add *at least* the URI into this tuple, making it a 4-tuple *(User,Host,Service,URI)*. It might be necessary to add some more data if we decide to interpret URI's in the rule as *prefixes*.

3.6 SSSD

System Security Services Daemon (SSSD) is a system deaemon intended for providing access to different remote authentication/authorization services consistently. It also provides caching [33]. To access different services, SSSD uses *provider plugins* specific to the identity management service.

SSSD has PAM (section 3.7) and NSS [61] interfaces [33] and also a public D-Bus [51] interface InfoPipe [34].

3.6.1 Architecture

SSSD is a daemon running on a machine that requests authentication/authorization, communicating with an identity management server. It consists of these main parts:

- *SSS client application* is the application that intends to use SSSD for authentication/authorization. It uses *SSS Client Library* to communicate with SSSD. For example, Apache HTTP Server using a SSS Client Library in its module 3.9.

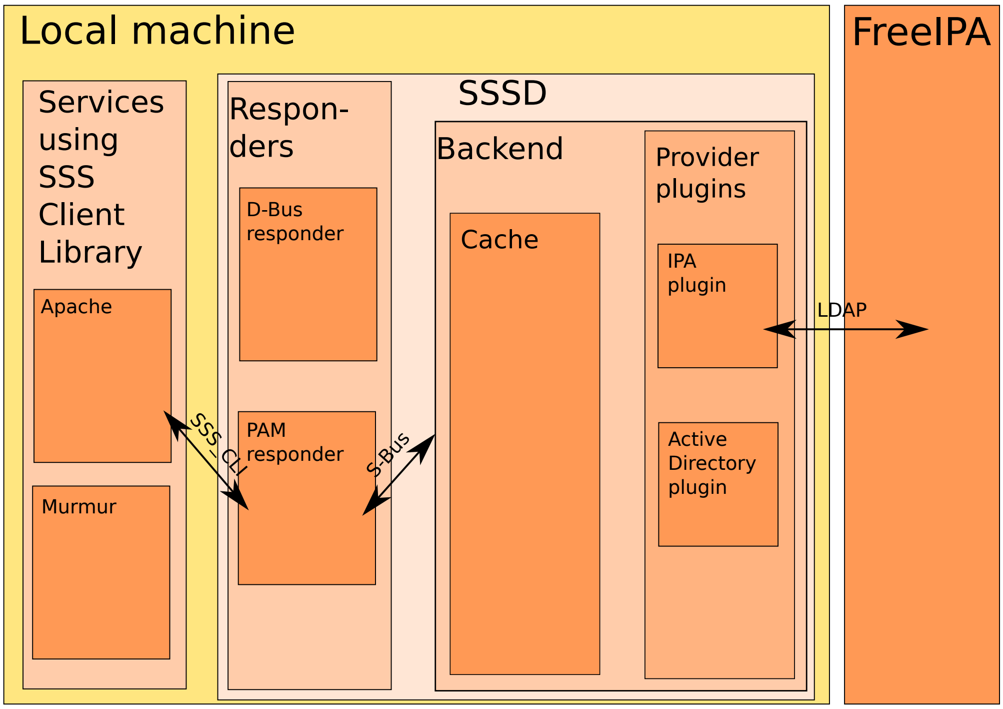


Figure 3.4: SSSD schema

- *Responder* is a process that communicates with the SSS client through SSS Client Library. They usually retrieve data from the Backend's Cache, if available, after checking they are current. Responders run in their own processes. For example, PAM responder or a D-Bus responder.
- *Backend* represents a domain and uses Provider Plugins to communicate with remote identity management services. Upon receiving a request from the Responder, the Backend performs communication with remote server, updates Cache and responds to the Responder. Each domain has its own process.
- *Provider Plugin* is a library used for communication between Backend and the identity management server. It is specific for the service it is communicating with. For example, IPA provider.

[35]

This schema is also shown in figure 3.4

3.6.2 HBAC rules caching and handling

HBAC rules are addressed by the IPA provider plugin. All the rules related to particular host and service are downloaded using LDAP and evaluated *on the SSSD side*. That means:

- More data than necessary might be transfered
- This data can be cached and SSSD can only download them from FreeIPA the first time for each host

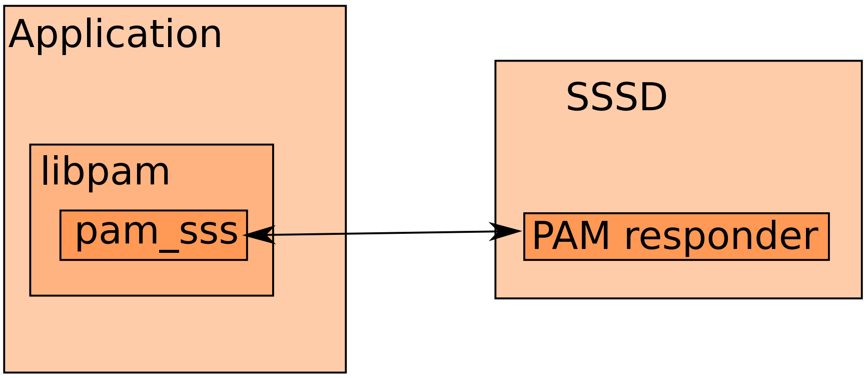


Figure 3.5: Communication between a PAM-enabled application and SSSD

After getting authorization request, SSSD updates the Cache if necessary (downloading information about host, service and all rules related to the host), and then, based on the cached data, responds either authorized or not-authorized.

3.6.3 InfoPipe

SSSD InfoPipe (or D-Bus responder) is a responder for communication between applications and the SSSD backend using D-Bus. It is in a separate package, `sssd-dbus`. It solves a problem with reaching additional information about entities in FreeIPA. Instead of applications connecting to FreeIPA directly, they can use SSSD which has acces to this information already. It uses both cache and online-lookup [8].

It does not allow access to all the data, instead, it provides object-oriented access to only users, groups, services and domains [9]. To use it for user authorization, we would need to add this functionality.

3.7 PAM

PAM is a unified way to authenticate/authorize users. Rather than implementing different identity management providers, the verifying application uses PAM as an interface. That allows applications to be independent on the underlying authentication/authorization scheme [58].

The application uses PAM library [24] (`libpam`), using header file `security/pam_appl.h`. The library invokes a PAM module. A PAM module is the implementation of some authentication/authorization schema and is selected based on PAM configuration files. Modules can also be stacked one on another. One of these modules is a *pam_sss* module allowing to authenticate/authorize based on information from SSSD (shown in figure 3.5).

Both D-Bus and PAM can be used by application to communicate with SSSD. While PAM is easier to use for its higher level of abstraction, the PAM API is very rigid. While there are `pam_get_item` and `pam_acct_mgmt` functions in the PAM API, they might not allow us to retrieve authorization information based on URI. It would also probably be a problem to change a very widely used PAM API. It may be more acceptable to modify InfoPipe's behavior than PAM API.

3.8 D-Bus

D-Bus is an inter-process communication mechanism. Its basic unit is a message including both metadata and data. It is binary and typed. Processes can talk one-to-one or to multiple processes. Processes can listen to events on the bus without being contacted directly. D-Bus messages are sent to objects which are addressed by their path names. Messages can be of multiple types, like signals, method calls, etc. The messages are very simple, there is no communication but one single message. They can request some action, data etc. If some return value is requested, it gets sent by another message. Data can be of a limited number of types resembling C types [51].

Objects are identified by:

- *Bus* identification; only one per bus and there is usually only one bus for application lifetime

- *Well-known name* of the service. It is a string consisting of lower and uppercase characters, separated by at least one dot. It should be unique, it is recommended to start it with a reversed DNS domain of the owner. For example, `org.freedesktop.sssd.infopipe.Users` is a valid well-known name.

- *Object path* within the service. It is denoted like a classical Unix-type system file path. There may be multile objects withing one service. For example, `/org/freedesktop/sssd/infopipe/Users`.

- *Interface* used. It specifies the method calls, their parameters etc.. It uses the same syntax as a well-known name. In smaller applications (like SSSD), it is ususally the same as the well-known name [4].

- *Member name*, which is the method to call / signal to emit.

There is a C D-Bus API. The header file to be included is `dbus/dbus.h`. There are multiple functions of sending, receiving, broadcasting and listening to signals.

3.9 Apache HTTP Server and modules

The Apache HTTP Server project is the most widely used web server daemon [47]. It is also our service of choice to show the functionality of our URI-based access control end-to-end, i.e. we are going to implement the functionality of a URI-based-access-control-aware FreeIPA client to the Apache HTTP Server. Due to architecture of Apache HTTP Server, it is not necessary to do any changes in the core project (which would probably not be accepted anyway). We can, instead, write a completely new *Apache module* to add this functionality when required.

3.9.1 Apache modules

Apache modules are pluggable objects used to extend Apache HTTP Server's functionality. They get executed using *handlers*. A handler is called on specified event. For example, an initialization handler called `child_init` is called on initialization of child Apache processes. If there is no module handling an event to enhance or change Apache's behavior, Apache's default handler is used. Modules are often written in Perl or C.

The handlers occur in the Apache main loop in different phases of HTTP request handling, as shown in figure 3.6. They are the following [40]:

- *URI translation* phase evaluates what the request is for
- *Access control* phase evaluates whether connection from the origin of the request is allowed
- *Authentication* phase authenticates the user
- *Authorization* phase determines whether the authenticated user is allowed to view the resource
- *MIME checking* phase determines how to handle the file requested
- *Response* phase sets HTTP headers and serves or interprets the file
- *Logging* phase logs the transaction
- *Cleanup* phase gets rid of resources allocated during request handling

Apache handler is a Perl or C function returning integer (a result of the operation). It can also change environment variables that can be available in later phases or they can issue internal requests, such as `internal_redirect` which processes another request instead of the one the module is called from. Handlers can even abort request handling completely.

The handler function is called with a single parameter, the *request record* (`request_rec`), that contains all the Apache's information about the transaction currently known, in a structure. Based on information from the request record, the handler does some action and when it is finished, it returns some status code.

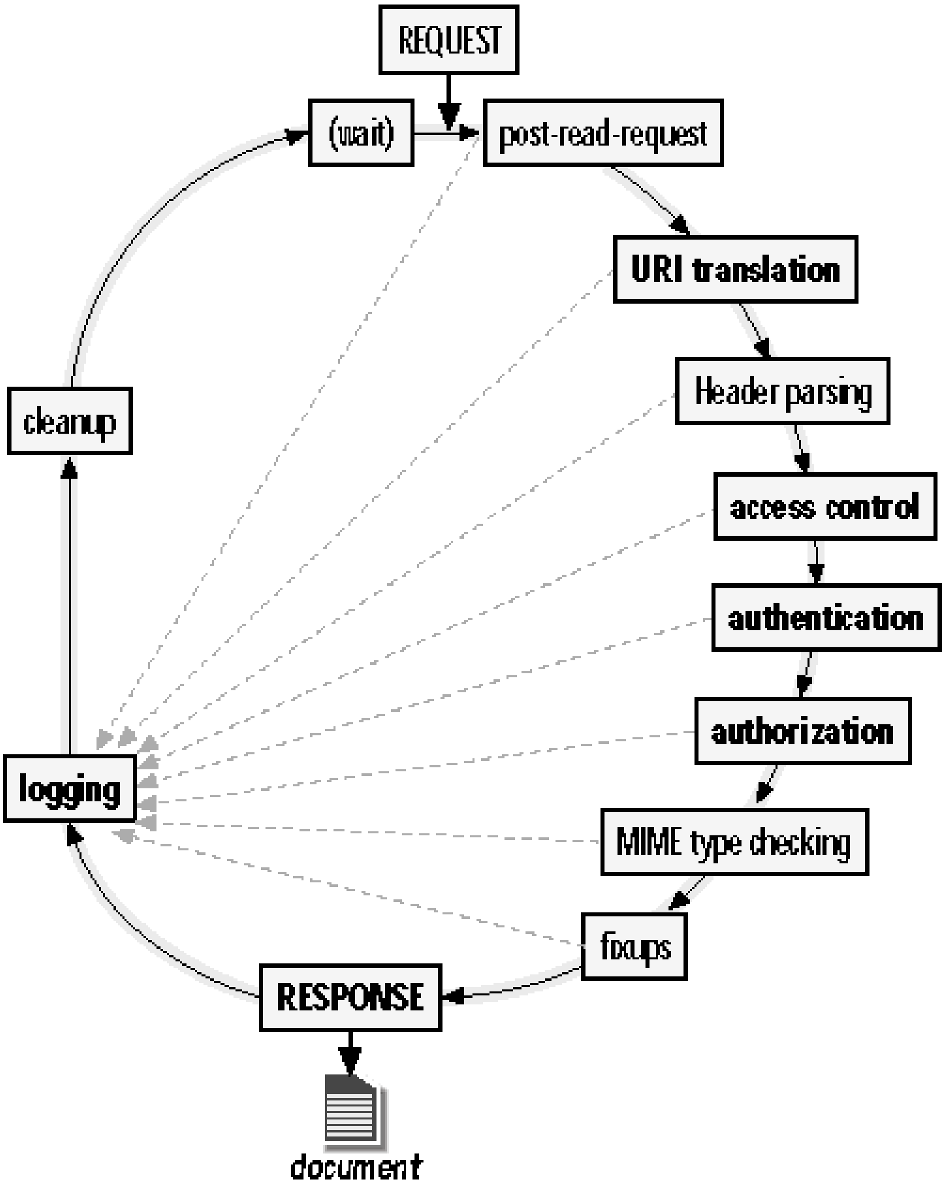


Figure 3.6: Apache main loop – an Apache module can have a handler for each phase in bold [40]

Chapter 4

Requirements specification

As we justified in the introduction, there is a need for a solution that can serve as a global access and authorization manager. It should be widely accessible and supported. The tool should be able to authorize authenticated client accesses based on URI of the resource on a given host.

With knowledge of specific requirements and global community expectations, we can specify our requirements more exactly. As first step, we will describe typical use cases. After that, we need to define what the improved system needs to be capable of and how it is expected to be used. We can also specify a typical installation and environment to assert our performance and optimization requirements.

4.1 Use cases

First, let us think about what the basic purpose of the RFE and the global community expectations are. Let us describe the reasons for RFE, current ways of substituting this functionality and typical use cases.

Multiple people in the Trac ticket have mentioned their need to use HBAC to authorize access not only by a 3-tuple *(User,Host,Service)*, but also by URI of the requested resource. They need to be able to set the rules centrally, using their current environment. That means the rules should be stored on FreeIPA server's side and modifiable through FreeIPA command line tools and WebUI. They need to keep the current infrastructure, the performance requirements should not be much higher than until now. They need to keep the same client tools if possible. They need to be able to decide whether user is allowed to access some URI on some service on some host. A useful property of this URI-based HBAC would be interpreting the URI as prefix, not merely testing for equality, as shown in the following use-case.

4.1.1 Authorization

A typical use case is a webserver service `W` on host `H`. The host is a client of a FreeIPA server `F`. An user `U` wants to access `W`'s resource identified by URI. `W` needs to decide whether to allow access or deny it, based solely on information from `F` (without using any local data, without running anything on the level of web application on `W`).

The mentioned entities are part of a big, enterprise-level infrastructure. The decision must be almost immediate as authorization will often be used in interactive applications.

The FreeIPA server must, after addition of this functionality, still be able to evaluate rules in environments with thousands of users and systems.

4.1.2 Administration

The decision from previous section should be based on some rules defined on FreeIPA side. These rules should be easily added, modified or deleted by current tools – the console tools *and* WebUI. This should be possible whenever a user is logged in with account that is allowed to do that.

4.2 Functional requirements

Based on the above, we can specify the requirements on functionality more formally:

- HBAC rules include information about URI
- The URI part of HBAC rule is modifiable by the same means as the rest of HBAC rule:
  - Command line tools
    - Add HBAC rule with certain URI
    - Change URI in HBAC rule
    - Delete HBAC rule including the URI part
  - WebUI
    - Add HBAC rule with certain URI
    - Change URI in HBAC rule
    - Delete HBAC rule including the URI part
- Allow read access to the URI part of HBAC rule at least by:
  - Command line tools
  - WebUI
  - LDAP
  - Other tools using the above
- Allow caching and evaluating by SSSD
- Do not raise performance requirements significantly for:
  - FreeIPA host
  - FreeIPA clients
  - The domain network infrastructure
- Do not raise significantly:
  - Time to allow access
  - Time to deny access
- Keep backwards compatibility:

  - When URI is not manually set, the default authorization behavior should be the same as it used to be
  - WebUI does not require the administrator to enter URI
  - Command line tools do not require the administrator to enter URI

- Works for at least the Apache HTTP Server and can be further developed for other services

4.3 Technological requirements

We defined what FreeIPA should be capable of after the improvement, let us define some technological requirements.

At this point, it is clear we are going to change at least one existing project, potentially more. These projects are in use for a long time by many users. We must not “break” them in any way, e.g. narrowing a set of platforms they run on.

The projects are open-source and must remain that way. This has practical reasons and as we need our solution to be accepted by upstream, this is really the only way. Another reason for open source base is also the fact that for a security-targeted application, it is beneficial (and, for many users, crucial and essential) property of the application to be open source [57]. It means higher level of trust of the user to the application as they can examine their source code to learn how it works and ultimately even verify if there are backdoors, given the user is willing to compile the application on their own. It also brings an opportunity to easily (compared to closed source software) further adjust the application to user’s needs.

Based on the above, we specify the following technological requirements:

- FreeIPA and any other potentially involved software must run on the same platform as it used to.
- FreeIPA and any other potentially involved software must run in the same environment as it used to.
- Newly developed software must run on the same platform as software it is by design expected to run with on the same host. We will perform tests at least on Fedora 23 with Apache HTTP Server in version currently in the Fedora 23 repository.
- All the software must be (and remain) open-source.

Chapter 5

Existing solutions

In the further text, we will analyze possible existing solutions or workarounds for authorization based on URI, specifically for a webserver. There are multiple partial solutions, they are, however, not general enough, or would break some standards or interfaces, or are plainly not based on identity management (i.e., they are too local).

5.1 Web application level authorization

We could, of course, just resign on any service-level authorization against some identity management server. The user would be either authenticated locally or against the identity management server. With this authenticated user's identity, we could decide, based on local information, whether the user is allowed to connect the required resource.

This is, in fact, a valid approach, but it is not the solution of our problem. Actually, this approach is complementary to the solution we are seeking for:

- Security should always come on multiple levels. Both on service level and on web application level, as mentioned in the introduction.
- This approach is capable of deciding with greater granularity.
- The application may consider the facts not globally stored on the identity management server.
- The application may consider the facts that are only specific for that single application.
- The application may further limit user's access rights. That means, even if the service allows the user to connect certain URI, the application might decide not to allow him, based on some fact mentioned above. It could even display some better explanation of why the user is not authorized to perform the action, as opposed to plain "access denied" from the Apache HTTP Server that would abort the request before the web application gets any chance to change something.

That being said, the approach of handling authorization on the web application level is valid, but is not the solution we are looking for. It is further good to mention it might be better not to handle authorization on the web application level *only*, despite it being a wide spread practice.

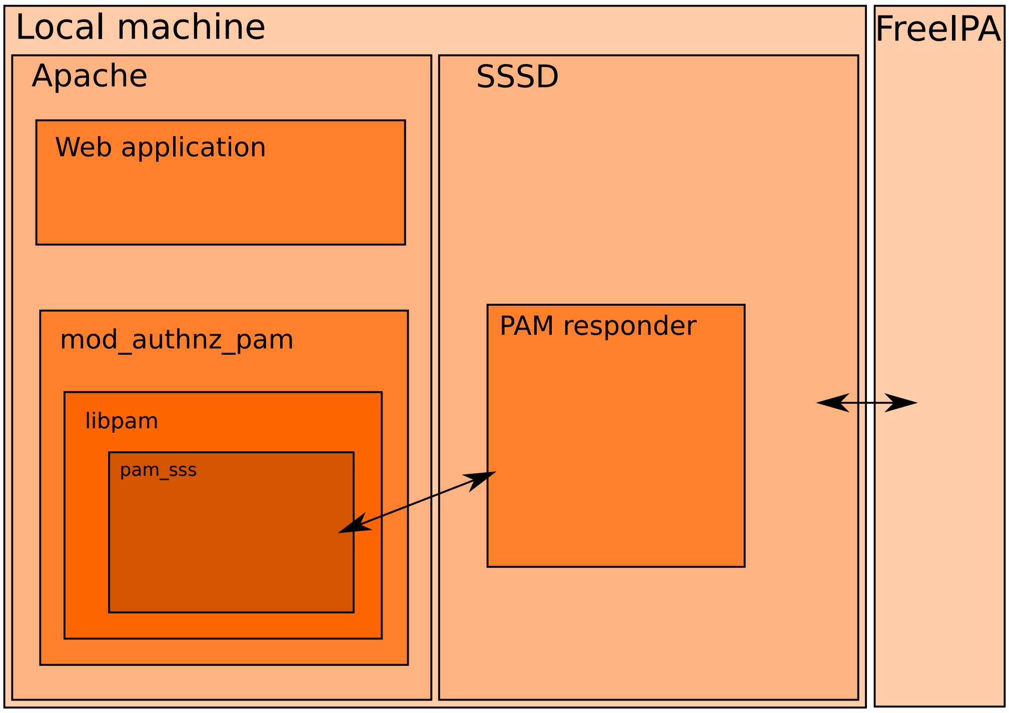


Figure 5.1: `mod_authnz_pam` usage

5.2 mod_authnz_pam

There is already an Apache module that handles authentication/authorization called `mod_authnz_pam` [7]. It can communicate with FreeIPA through SSSD. Communication with SSSD is done via PAM. The whole Apache ↔ FreeIPA communication is shown in figure 5.1.

The `mod_authnz_pam` module is a small-scale tool that would be easy to change. It should also be possible to do changes to SSSD so that it allows authorization based on URI. The only changes required in FreeIPA would be storing URIs in the rules as the evaluation of these rules happens on SSSD's side already.

As mentioned, however, the module uses PAM to communicate with SSSD. And as mentioned in section 3.7, the PAM protocol is not easy to be changed. Many applications rely on PAM API's consistency and we would need to find a way to communicate some additional data, like URI, without doing changes to this API. This is a complication while extending `mod_authnz_pam` to be URI-aware.

That being said, this approach is interesting to work with, but PAM API is a big limitation.

5.3 mod_lookup_identity

Another interesting Apache module is `mod_lookup_identity`. It is able to retrieve some additional information about the user.

The module has nothing to do with authorization. It is, however, interesting for its way of retrieving the information. It communicates with SSSD through InfoPipe 3.6.3. As we mentioned in the section about InfoPipe, however, InfoPipe only allows getting information about certain elements, none of which is a HBAC rule. If we wanted to use an approach similar to InfoPipe, we would need to do one of the following:

- Add the ability to list HBAC rules (all or related to the host) to InfoPipe and handle the rule evaluation on Apache module's side. That would be a less general approach as every service would need to handle this on their own, our solution would be only useful for services which have a module similar to our new Apache module designed.
- Add the ability to authorize user over D-Bus. We could perhaps consider this a user's property – thus using existing User interface and extend it with a function evaluating authorization based on 4-tuple *(User,Host,Service,URI)*. That would mean only the yes/no answer would ever be responded to the query over D-Bus. Implementing a function that merely asks for authorization over D-Bus would be easy.

Using some module similar to `mod_lookup_identity` seems like a feasible approach, despite the module itself being unable to solve our problem.

5.4 Changing PAM service name based on location in Apache HTTP Server

It is possible, in Apache HTTP Server's configuration files, to divide a web application into *locations*. These locations are based on URI prefix. It is possible to specify a distinct PAM service name for each of the locations.

That would allow us to allow the user to only access certain parts of the web, depending on URI. The available part would be in one location and the part which can not be accessed would be in another location. There might be multiple locations if there needs to be more fine-grained distinction, for example if there are multiple user groups with distinct access rights. For example, we could do the following:

- Map the `http://web.net/users` URI prefix to location with PAM service set to `http-users`
- Map the `http://web.net/admin` URI prefix to location with PAM service set to `http-admin`
- In FreeIPA, allow user `U` to access this host's service `http-users`
- In FreeIPA, *do not* allow user `U` to access this host's service `http-admin`

This way, the user `U` would be able to access URIs with prefix `/users`, but not `/admin`, which is the URI-based authentication we want. However, while the data in FreeIPA is indeed managed by the identity management service, the data in an Apache HTTP Server's configuration file is purely local. In this case, the result would be FreeIPA knowing it does not allow connections to the service named `http-admin`, but having no information whatsoever about the relation between URI and the service name; every service in the domain could have different local mapping and would need to specify it, which would effectively degrade this solution to local access management.

Chapter 6

URI-based access authorization in FreeIPA

In the previous chapters, we have thoroughly described the requirements on the results of this work. We have summarized our motivation to engage in it. Thus, we know the intended purpose of the new functionality and we know the target users and system and platform specifications.

We have shown the technological background and described terms we are going to operate with further in this text. We have mentioned multiple applications, protocols and standards and shown how they make sense together. We also demonstrated the current solutions and shown that they are either not sufficient or useful for our specific purpose, and why.

In this chapter we will come with a concept of our own enhancements to current solutions in order to support the requirements defined.

6.1 Current state

Currently, neither FreeIPA, SSSD or Apache HTTP Server is aware of authorization's relation to URI of the requested resource. FreeIPA can not store the information necessary in its LDAP backend, SSSD can not either store it in its cache nor can it get this data from FreeIPA, and there is no Apache module that takes URI of the required resource into account.

There are multiple possible workarounds, most notably the one described in section 5.4 – that would not even require changes in the projects involved, only changes in the Apache HTTP Server's configuration files would be necessary. Neither of them, however, fulfils all the requirements and they do not really solve our problem.

6.2 Enhancement of FreeIPA and related applications – possible approaches

Now that we have defined what needs to be done and why it can not be solved by current tools, let us show some ways the problem could be solved:

6.2.1 Using a Kerberos Authorization-data field

As we mentioned in section 3.3.3, there is a field in a Kerberos ticket called "Authorization-Data". This field is optional and the standard does not define what it is supposed to contain.

This field could be used to store some data about what actions the user is allowed to perform. It could contain information about all the URI prefixes the user whose ticket is being used is allowed to access. This approach would have some benefits:

- The service would not need to be IPA-aware at all; mere Kerberos-awareness would be enough
- There would be almost no changes in the service necessary; it would have all the information it needs and would only evaluate whether the requested URI matches some of the listed prefixes

This approach, however, has some serious problems:

- The access rights are not, in FreeIPA schema, a property of the user. They are derived from the HBAC rules.
- There is a big potential of colliding with some standard/expectation. While contents of this field is not specified by the Kerberos standard, both MIT and Microsoft Kerberos implementations use this field for their own implementation-specific data. Even if some service would not use this data and therefore would be capable of using our newly defined data, some other service might rely on this Kerberos implementation-specific data. The solution would not be universal and might potentially break a lot of existing infrastructure.
- This approach would not be consistent with current approach to evaluating HBAC rules and authorization at all. So far, FreeIPA uses another approach described in section 6.1. While the current state would theoretically be achievable by the approach of putting the authorization data in the Authorization-Data field in the Kerberos ticket, the FreeIPA engineers decided not to. It would be wise to use and enhance their current approach instead of redefining HBAC architecture completely.
- The evaluation would need to be on both sides – the service and FreeIPA. As access rights are not a property of the user, FreeIPA server would need to, upon request, look at all the HBAC rules, from these, make a set of URIs related to the user that is being authenticated/authorized while connecting to the verifying host's service, and put this set to the Authorization-Data field of the Kerberos ticket. The service would then need to match some of the allowed URI prefixes to actual URI being used, or decide that none of them matches.
- There might be no Kerberos ticket at all during authorization – authentication could have happened by method other than Kerberos.

6.2.2 Using LDAP directly

We could instead design an Apache module that would have an authorization handler. This handler would connect the FreeIPA's LDAP directly (figure 6.1). After contacting LDAP,

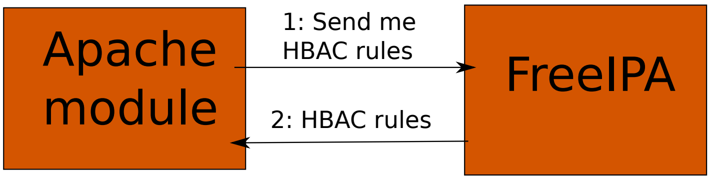


Figure 6.1: FreeIPA contacted over LDAP

it would download all the HBAC rules related to the user, host and service (the Apache HTTP Server itself) being run.

As this data would be accessible to the module, it could evaluate the rules on its own, deciding whether any of the URIs in the rules are prefixes of the URI of the resource requested. If they were, the access would be allowed; if none of them matched, the access would be denied.

The benefits of this approach would be:

- This approach would be very easy to implement. It would be just an Apache module hook connecting to LDAP directly and acting upon received data.
- The service would have full control over any thinkable aspect of the problem. It would have full information accessible.

The problems of this approach would be:

- A lot of data being transfered more times than necessary – the module would not cache anything and would require all the data needed to evaluate the rules being sent. If we actually tried to implement this functionality to counter this effect, we would be re-inventing SSSD.
- While this approach is possible and easy to implement, it has no other real advantages over the approach explained in section 6.2.5.

6.2.3 Using FreeIPA directly

Another possible approach is making FreeIPA itself evaluate the HBAC rules and merely respond with "authorized" or "not authorized" answers to the service that is asking (figure 6.2). That would require using some FreeIPA public API. The authorization function would be called by a remote service with User, Host, Service and URI as parameters. It would go through HBAC rules, try to match them and if some of them matched, allow access. The service-side functionality would be mere asking and receiving a boolean answer. Therefore, the good thing about this approach would be easy extendability of other services by this functionality. The bad properties of this approach would, however, be:

- Evaluation would happen on FreeIPA's side. That would mean heavy load in bigger domains with many services that are being accessed frequently.
- It would not be possible to cache anything other than response for the exact input parameters because the application would only receive binary authorized/unauthorized response.

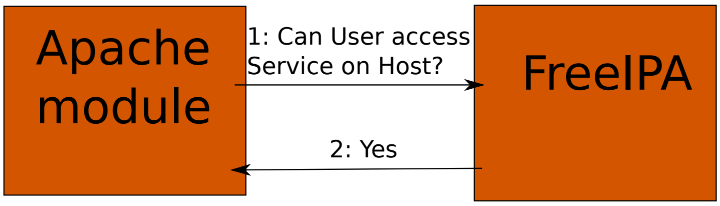


Figure 6.2: FreeIPA contacted directly

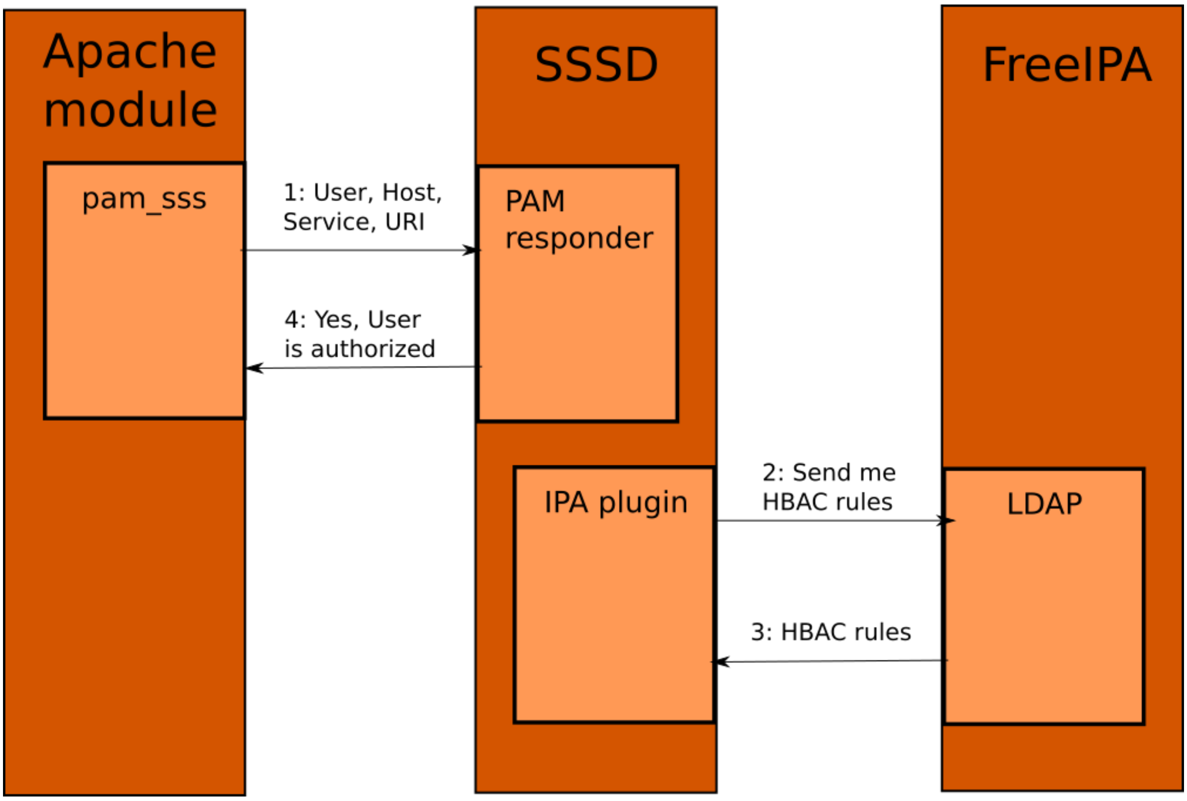


Figure 6.3: SSSD, PAM

- We would need to always ask FreeIPA for authorization. The request would go through network (as FreeIPA will be almost always on a remote system) and that would be a big problem, especially for interactive applications.

6.2.4 Using SSSD over PAM

In another approach, we would use SSSD to evaluate HBAC rules it gets from FreeIPA (which is the solution currently frequently used for authorization) and Apache modules would ask SSSD for authorization using the PAM library (figure 6.3).

Here, FreeIPA would merely store and manage the HBAC rules, not evaluate them. SSSD would download them from the FreeIPA's LDAP backend and cache them. The service (e.g., Apache HTTP Server) would contact SSSD and SSSD, upon receiving the necessary information (User, Host, Service, URI) would evaluate the cached HBAC rules and if any of them matches, allow access; if none of them matches, deny it. The response would be a mere yes/no response. The request would be repeated many times as the response would not be cached and the service would always only receive this boolean response. It is, however, not a problem as SSSD is supposed to run on service's local system. This approach is also the current way of handling HBAC.

Communication between the service (in our case, the Apache module) and SSSD would happen over PAM in a way similar to how `mod_authnz_pam` described in section 5.2 works. It might even be a good idea to just use this module as a basis and improve it to be URI-aware. That would require using PAM to somehow send additional data to the SSSD: the URI. We would also need to extend a PAM module used for communication with SSSD's PAM responder – `pam_sss`, to send this data further to SSSD. After that, SSSD would evaluate the request as described in the previous paragraph and respond yes/no – this response would eventually get through PAM back to the Apache module.

This approach has multiple advantages:

- A big part of it is already part of FreeIPA. We already have the HBAC infrastructure, it is just not aware of resource URI.
- Very easy to implement in the service – it only needs to issue a PAM request with necessary information and receive the yes/no response, it does not evaluate anything.
- For larger-scaled domains with multiple frequently accessed services, the FreeIPA server's performance will not significantly drop compared to the current state. There is no evaluation taking place on FreeIPA's side.

Compared to the approach described in section 6.2.3 (getting the yes/no answer directly from FreeIPA), this would generate more network traffic as SSSD would download all the related data needed for evaluation of the request as opposed to only receiving a boolean yes/no answer.

6.2.5 Using SSSD over D-Bus

The last proposed solution is very similar to the approach described in the previous section 6.2.4. The FreeIPA part would be absolutely the same. Also, SSSD part used for communication with FreeIPA would not need to be changed comparing to that previous proposal.

However, communication between the service (in our case, the Apache module) and SSSD would happen over D-Bus (figure 6.4) (in contrast to way `mod_authnz_pam` described in section 5.2 or the solution proposed in the previous section handles communication with SSSD – over PAM). That means the Apache module would need to use D-Bus (as opposed to merely calling a PAM library), and SSSD would need to understand the data on D-Bus as request for authorization and handle it and reply accordingly. SSSD's D-Bus Responder is Infopipe (described in section 3.6.3). InfoPipe does *not* currently have such a call and does not even allow access to HBAC rules, that would need to be implemented. While addressing a different problem, the approach described in section 5.3 is technically very similar in its part related to communication between Apache HTTP Server and SSSD.

This approach has multiple advantages, very similar to the one using SSSD over PAM:

- A big part of it is already part of FreeIPA. We already have the HBAC infrastructure, it is just not aware of resource URI.

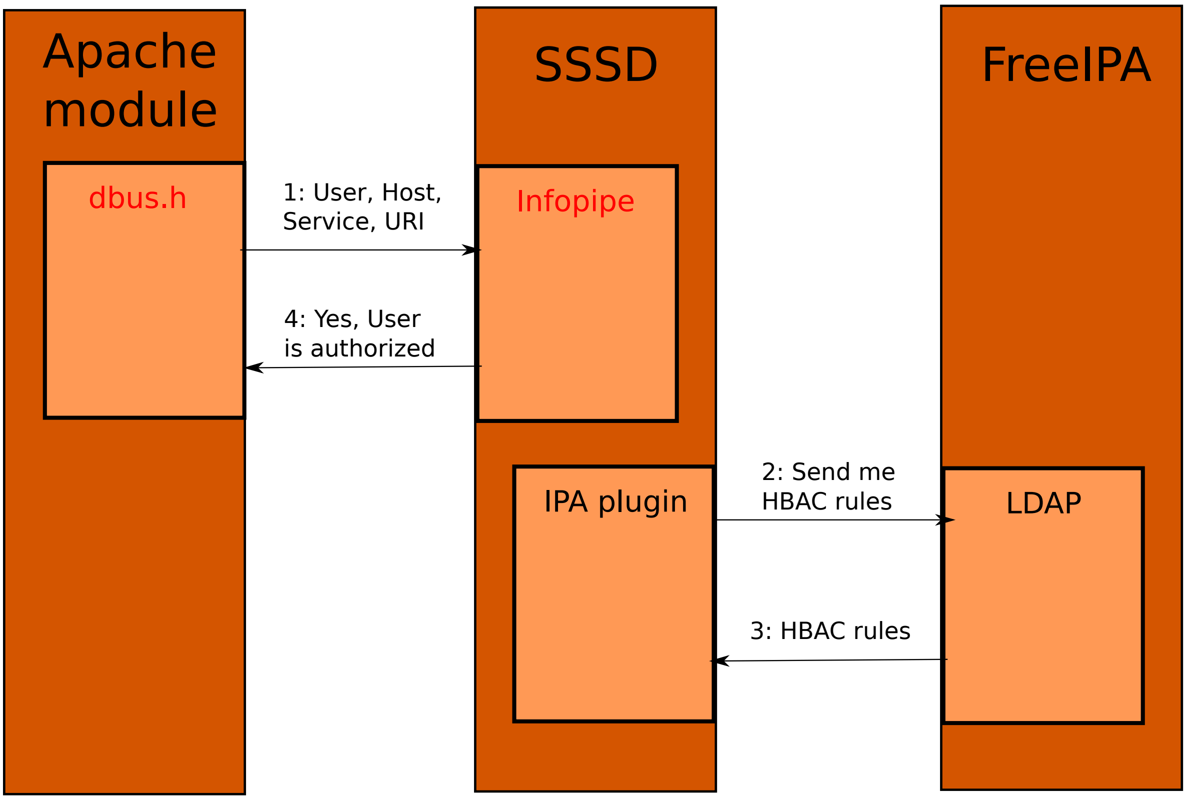


Figure 6.4: SSSD, D-Bus

- Very easy to implement in the service – it only needs to send a D-Bus request with necessary information to SSSD and receive the yes/no response, it does not evaluate anything. However, this is still harder than merely calling a PAM library function.
- For larger-scaled domains with multiple frequently accessed services, the FreeIPA server's performance will not significantly drop compared to the current state. There is no evaluation taking place on FreeIPA's side.

Compared to the approach described in section 6.2.3 (getting the yes/no answer directly from FreeIPA), this would generate more network traffic as SSSD would download all the related data needed for evaluation of the request as opposed to only receiving a boolean yes/no answer.

6.3 Proposed approaches comparison

While all of these options are possible, they all have certain drawbacks. Some of them are so dire that we can immediately decide not to use these solutions if we find better ones:

- Firstly, we can reject using Kerberos Authorization-data field. While it *might* be possible to communicate authorization data using Kerberos ticket, *there might be no Kerberos ticket at all* while we are authorizing a user. We merely expect *some authentication* has happened before authorization, but some other means might have been used.

- Also, while using LDAP directly would be possible and easy to implement for sure, there are performance issues (all the HBAC data would need to be downloaded on each request) and it is not consistent with any of the approaches already used.

- While using FreeIPA directly is possible, for larger environments, this would mean excessive load of the FreeIPA host. Also, this approach is never used, canonically, SSSD is used for both evaluating and caching rules.

The other two solutions are both suitable. They use SSSD which is a good and used practice, they solve performance problems, they partly use existing infrastructure.

They are very similar and the difference lies in communication between Apache module and SSSD. One would require the Apache module to use PAM and to extend `pam_sss` and SSSD's PAM responder, the other would require module to use D-Bus directly and extend Infopipe to understand it.

Standard PAM does not use URI item in its request, however, it is possible to use environment variables. Infopipe does not currently allow access to HBAC rules, but it can be extended. From the two almost equally suitable solutions, we choose the one described in section 6.2.4 where FreeIPA is used to store authorization information, SSSD evaluates it and Apache module communicates with SSSD using PAM, mainly for ease of implementation and its usage of a wide-spread PAM interface.

Chapter 7

Concept

After mentioning and comparing multiple possible approaches, we decided to use the solution described in this chapter. It is one of the two feasible solutions and while another one is also a valid option, the chosen solution is more consistent with existing tool usage, easier to implement and more reusable thanks to usage of PAM.

The high level architecture is shown in figure 7.1. It shows different parts of the infrastructure in context of URI-based HBAC:

- FreeIPA, the identity management server, as a part for storing and manipulation of HBAC rules. Adding, deleting and modifying HBAC rules can be done both by means of WebUI and command line. The data is stored in FreeIPA's LDAP where it can be accessed either by FreeIPA's tools or directly using LDAP tools or libraries, limited only by access rights of the requesting LDAP user.

- A link between FreeIPA and SSSD. SSSD uses pure LDAP to get HBAC rules from FreeIPA. That means FreeIPA is not the part making any decision in phase of using/evaluating HBAC rules in this schema and only serves for their management.

- SSSD side, consisting of these important parts:

  - IPA provider plugin which is the part that actually decides whether user is authorized to access based on (user,hostname,service,URI). It uses SDAP [32] ("SSSD LDAP") plugin to get this data from FreeIPA which in turn uses LDAP plugin. IPA provider plugin receives authorization requests from PAM responder through S-Bus (SSSD's partial implementation of D-Bus). Based on information specified in this request and information received from FreeIPA, it decides whether to allow access or not and responds back to PAM provider, also through S-Bus.

    A benefit of evaluating rules on SSSD's side is less load of the FreeIPA server. SSSD caches the HBAC rule data so there is almost no increase in data traffic.

  - A link between the IPA provider and PAM responder, S-Bus.

  - PAM responder which is a tool listening for PAM requests on a `AF_UNIX` socket. When it receives a request from some application, in our case some application using `pam_sss` library, it forwards this request to IPA provider plugin, waits for its response and responds back to the application.

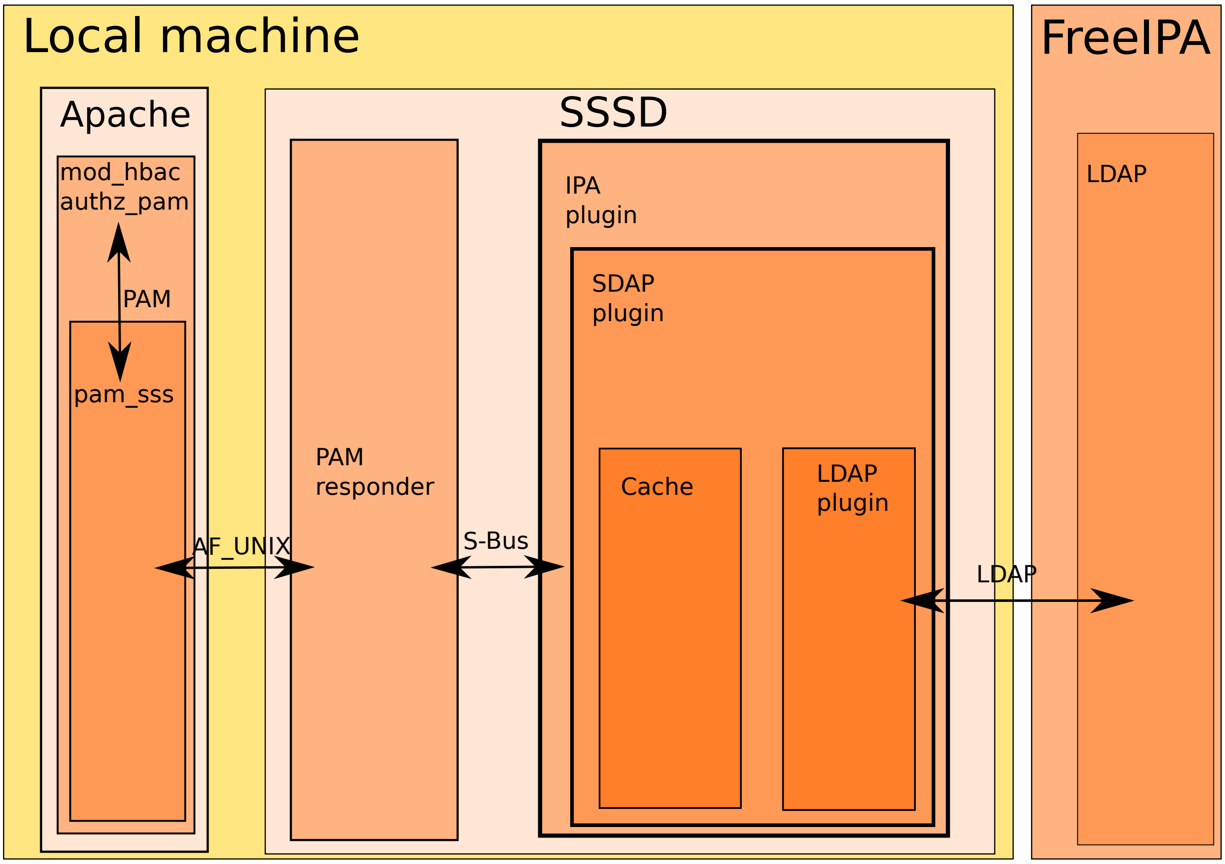


Figure 7.1: High-level architecture – depth levels denote the “higher using lower” relation (as a library), arrows denote inter-process communication (protocol shown above the arrows).

- A link between SSSD and application, `AF_UNIX` socket. It is used by `pam_sss,` the applicaiton does not need to be aware of this communication as it is encapsulated by PAM interface.

- The application side – the service requesting authorization. Specifically, Apache HTTP Server using the `mod_hbacauthz_pam` module which is using PAM interface, with `pam_sss` being specified as provider.:
  - Apache HTTP Server is the service we are going to use as a an example as it is one of the most basic and known services identifying its content by URI. Furthermore, web applications have some specifics that are important to consider while thinking of a suitable strategy of evaluating HBAC rules. We will create an Apache HTTP Server authorization module which will have the role of application requesting user authorization based on HBAC.
  - `mod_hbacauthz_pam` is the module using an Apache HTTP Server hook for authorization. It uses PAM to communicate with SSSD using `pam_sss` library.
  - `pam_sss` is the library which is part of the SSSD project and joins the PAM interface on one side with SSSD's PAM responder on the other side using AF_UNIX socket.

Together, these tools are a way to store and manipulate HBAC rules in FreeIPA, request authorization based on these rules in some applicaiton (namely Apache HTTP Server) and evaluate the rules to get the answer.

This high-level design has multiple advantages:

- Utilization of existing infrastructure to achieve new functionality. Many parts can be reused and there are standardized interfaces used.

- Very easy implementation in applications. The application does not need to know the actual rules nor does it need to use some new protocol. The application merely needs to send an authorization request using standard PAM interface and receive the answer which is only *authorized* or *not authorized.*

- There is no evaluation happening on FreeIPA server – applying rules are evaluated on client side, in SSSD's FreeIPA plugin. This is important in environments with many clients.

- There is no excessive data usage – only rules that have chance to be matched are transfered to SSSD and they are cached. They are, however, checked for change to keep them up to date.

Before describing implementation, we will show a concept of some important parts of the schema in detail and justify some choices that have been made.

7.1 Conceptual problems and solutions

We need to implement a way to store and manage URI of the resource in FreeIPA. We will add it as an attribute of already existing entity: HBAC rule. We also need to add URI evaluation functionality to SSSD, enhance `pam_sss` to handle URI data and create a sample client application using this functionality.

There are multiple decisions that need to be made. Mainly format of the URI – what exactly it should express. There are also concerns with case sensitivity of the URI, which parts of the URI exactly should be stored, and what is the relationship to other attributes of the HBAC rules. Backwards compatibility is another important issue that needs to be addressed.

7.1.1 Semantics of URI in HBAC

URI is another attribute of the HBAC rule. An attribute (or group of attributes) is a way to constrain a set of requests the rule applies to. If *all* of the attributes are met, then the HBAC rule is considered *matched*. These attributes are: service (service group), hostname (host group), user (user group), and URI.

That means, even if user, hostname and service match, we have to decide whether URI matches or not – if it does, the whole rule matches, if it does not, the whole rule does not match for sure. There are multiple ways of interpreting URI attribute and the right choice is not obvious – it is important to keep in mind that URI has some properties that change throughout different applications, e.g.:

- Some services consider URI case sensitive, some do not.
- Some services have hierarchical URIs: parts are often divided by slash. This is not, however, by definition of URI.
- Often, each part of the hierarchically-interpreted URI means more and more strict access rules to the resource specified by that URI. This is the case of web applications, for example. Some applications, however, might use completely different addressing structure.

Case sensitivity

Case sensitivity influences whether certain URI attribute, whatever its exact semantics is, is matched to requested URI or not. URI's case-sensitivity question is not trivial as there are multiple documents related to it:

Generally for URI, RFC-3986 [42] specifies in its 6.2.2.1 part that *scheme and host are case-insensitive* but *the other generic components are assumed to be case-sensitive unless specifically defined otherwise by the scheme*. RFC-2616 [46] in its 3.2.3 part (named *URI comparison* states that *a client SHOULD use a case-sensitive octet-by-octet comparison*, and further enumerates exceptions, some of these being that *comparisons of host names MUST be case-insensitive* and *comparisons of scheme names MUST be case-insensitive;*. RFC-2616, however, speaks about http client comparison – although in our case the URI comparison happens on client-side (meaning a FreeIPA client – SSSD), it does not happen on a *http client*. This means the rule does not really cover our case, it is just something that is good to keep in mind. Furthermore, RFC-7230 [56] in its part 2.7.3 again specifies that *scheme and host are case-insensitive* and *all other components are compared in a case-sensitive manner*.

From these, we can conclude we should compare scheme and host as case-insensitive and we should normalize them to lower-case [42][56]. All other parts should be left as they are and compared in case-sensitive manner.

Hierarchical interpretation of URI, typical address structure of web applications

In many applications, URI is structured hierarchically. While it is possible to have URI as a seemingly random string (except for scheme and authority parts), it is usually preferrable to give it some clearly-defined structure. In web applications, it is often so that parts of URI divided by slash symbol can be mapped to directories and the part after authority defines path from some root directory to the source of the page. In other cases, often when using some web frameworks, this part after authority can not be mapped to some path on filesystem, however, it is some logical path in the application/framework.

Typical for these "path" interpretations is that usually, the longer the URI is, the more specific part of the web is being shown. Consider this example:

```
http://hostname.net/app
http://hostname.net/app/users
http://hostname.net/app/users/user42
```

Clearly, the longer URI (with the same prefix) means more specific page. This is de facto standard in web applications and is rarely not the case.

Another typical thing is that *the more specific the URI is, the stricter access rights are expected to be set.* This is a very important finding because it has dire implications on how we can interpret the URI parameter in HBAC rule. There are multiple possible approaches and some of them might make sense for other applications, but if we consider web applications (and as we will see, this will lead us to more general approach), we neccessarily need to find out some of them are simply not feasible from web-administrator point of view and that they can not be used for general URI as they pose expectations on the URI structure that might not be always true. Consider this (web) application structure:

```
http://hostname.net/app/public
http://hostname.net/app/auth/user1
...
http://hostname.net/app/auth/user42
...
http://hostname.net/app/auth/admin
```

We see that the more specific page, the stricter the access rights should be, as long as prefix is the same. We want to set the following access rights:

- Anyone (even unauthenticated user) can access `http://hostname.net/app/public`.
- Any authenticated user can access `http://hostname.net/app/X` for `X` being any user's name except `admin`.
- Only admin can access `http://hostname.net/app/auth/admin` (so: while any user can access anything with prefix `http://hostname.net/app`, *only admin* can access `http://hostname.net/app/admin`).

As the number of users of the application might be very large, unknown, or infinite, it is not easy to find a way to interpet the URI so that it is not hard or impossible to come up with rules achieving the above mentioned goal. There are multiple ways of interpreting URI, the considered ones are the following:

- *Exact URI:* Two URIs match if and only if they are exactly the same. This is the simplest approach, but is very user unfriendly and most importantly, it is not even capable of achieving the goal for most of the dynamic web applications: they usually have dynamically changing number of URIs that can be legally accessed. In the example it would be sufficient to allow registration of new users to make this approach completely useless.

  How the rules would look (omitted `HOST any SERVICE any` everywhere):

  ```
  ALLOW any URI http://hostname.net/app/auth/user1/a
  ALLOW any URI http://hostname.net/app/auth/user1/b
  ...
  ALLOW any URI http://hostname.net/app/auth/user1/x
  ALLOW any URI http://hostname.net/app/auth/user2
  ...
  ALLOW any URI http://hostname.net/app/auth/user42
  ...
  ALLOW admin URI http://hostname.net/app/auth/admin
  ```

  There is no problem with giving some user too much access rights, but there are as many rules as there are URIs with prefix `http://hostname.net/app/auth/`.

- *Prefix interpretation allowing access when any rule matches:* As we see, the longer the URI, the stricter access control rules. This leads us to a concept of prefix-matching the URIs: whenever the URI in rule is a prefix of the requested resource's URI, the rule matches in terms of URI.

  The way HBAC rules are interpreted, however, is currently such that whenever *any* HBAC rule matches, the access is allowed. It is a correct behavior when only considering (user,service,host), but it causes a problem when trying to include URI as a matching parameter. As shown in the last item of a goal list, we need to have a way to allow every URI with certain prefix `A` *except* URIs with certain prefix `B` where `A` is a prefix of `B`. In other words, we need to exclude a subset from a set of URIs described by certain HBAC rule. This is not possible when matching *any one* rule causes access authorization.

  How the rules would look (omitted `HOST any SERVICE any` everywhere):

  ```
  ALLOW any URI http://hostname.net/app/auth/
  ALLOW admin URI http://hostname.net/app/auth/admin
  ```

  This solves the problem of too many rules. However, *it does not work*! While the second rule only allows admin to access `http://hostname.net/app/auth/admin`, the first rule allows any user to access *everything with prefix `http://hostname.net/app/auth/`, including `http://hostname.net/app/auth/admin`.* While the problem of too many rules is solved, there has arisen a new problem: we can accidentally allow access to larger set than intended and there is no way to set exceptions from that set. In this example, there is actually no way to set the rules correctly so they achieve the goal, except using every possible prefix *other than* the intended exception, effectively making this approach as bad as the first one: while it is not possible to have a rule for every possible URI with given prefix (in the example, we would not need the parts after usernames), there is still potentially infinitely many rules:

```
ALLOW any URI http://hostname.net/app/auth/user1
ALLOW any URI http://hostname.net/app/auth/user2
...
ALLOW any URI http://hostname.net/app/auth/user42
...
ALLOW admin URI http://hostname.net/app/auth/admin
```

- *Prefix interpretation utilizing DENY rules:* To solve the problem of exception from a set of allowed URIs, we could come up with a concept of DENY rules. The approach would mean allowing access when any ALLOW rule matches and *no* DENY rule matches. A DENY rule would otherwise be the very same rule as an ALLOW rule. That would not be completely new for FreeIPA – at certain point in time, there actually were both ALLOW and DENY rules.

  DENY rules were, however, dropped from FreeIPA. The reason for this is that we believe that access rules should always be described positively – listing all accesses that are allowed, rather than listing what is not allowed and thus risking we forget something or make a mistake that would allow access that should not be allowed. Another reason is that when we, for some reason, do not evaluate an ALLOW rule, the result is denial of service at worst, while failing to evaluate a DENY rule could allow access that should not be allowed. After consulting with upstream, it seems DENY rules are absolutely not intended to be added again.

  Furthermore, merely adding DENY rules would not be sufficient; for example, there would be no easy way to come up with rules for our example. We would need to deny access to `http://hostname.net/app/auth/admin` to large or infinite number of users as the access would be allowed by the first rule. The rules would look something like:

  ```
  ALLOW any URI http://hostname.net/app/auth/
  DENY user1 URI http://hostname.net/app/auth/admin
  ...
  DENY user42 URI http://hostname.net/app/auth/admin
  ```

  This could be solved by only matching the user-wise most specific rule or giving the rules some order, e.g.:

  ```
  1 ALLOW any URI http://hostname.net/app/auth/
  2 DENY any URI http://hostname.net/app/auth/admin
  3 ALLOW admin URI http://hostname.net/app/auth/admin
  ```

  This would be a fully working solution, allowing exceptions, describing infinite number of cases (both URI- and user- wise) in a relatively small number of rules, and relatively readable. Still, there are drawbacks:

  - It is not easy to determine a rule to compare which one of the rules is more specific user-wise. It would also be very error-prone.
  - Adding order to rules would mean a significant change in their semantics which would be hardly accepted by upstream.

  - DENY rules will probably never be accepted by upstream.
  - There are better and simpler solutions, described further in this text.

- *Longest-prefix matching:* Using the previous notion, we would in many cases create a pair of rules for subsets we wish to exclude some users from – an ALLOW rule allowing access to certain subset of users, and a DENY rule which is the same except it denies *any* user access to the same location (which is necessary in case there is an ALLOW rule allowing access to some URI which is a prefix of this location's URI). The more specific or latter of those (depending on which approach we would choose) two rules would be the ALLOW rule and the result would be only allowing access to that URI to certain users.

  In previous example, this exactly happens: rule 1 allows access to `http://hostname.net/app/auth` to anyone and to allow access to `http://hostname.net/app/auth/admin` to admin *only*, we first need to deny everyone access there by rule 2 before allowing it again for admin only by rule 3.

  It is easy to understand why the DENY rule could be there implicitly – when admin allows access to some resource to some user, he means that user *only* and all other users should be denied. However, there is another rule that allows access to anyone - the first one. To solve this problem, we can state that we only want to decide based on the rule with *longest prefix match*. Even if there are multiple rules matching, we are only interested in the most specific one. This allows us not to use DENY rules at all because when there is no ALLOW rule, access is denied implicitly, and the more general rule allowing access to a superset of the more specific rule would be ignored.

  When searching for the longest prefix, we should only take into account the rules that match in terms of host, service and schemeAndHost (section 7.1.2) because finding longest URI match usually makes sense in the same application and once allowed access should not be denied based on rule intended for another host or service. For example, when we allow anyone access to the webserver A with URI `/auth`, we do not expect another rule allowing `admin` access to FTP server B with URI `/auth/user` to also deny access to the webserver A with URI `/auth/user` to anyone but `admin`. The only attribute that can not match for the rule to deny access to the same URI for users not listed in it is *user* because only this attribute does not change the context of URI to some completely different application.

  In longest-prefix case, we could use the same rules as in the previous example, just ignore the ordering and drop the DENY rule (because there will be no DENY rules):

  ```
  ALLOW any URI http://hostname.net/app/auth/
  ALLOW admin URI http://hostname.net/app/auth/admin
  ```

  The first rule allows anyone access to URI's beginning with `http://hostname.net/app/auth`, *except for longer URIs which have URI `http://hostname/app/auth` as a prefix.* The first rule's URI is the second rule's URI's proper prefix, thus the first rule is ignored for any URI matching URI of the second rule, regardless whether the first rule's URI matches or not. This serves as implicit deny for everyone accessing the resource identified by an URI which is a prefix of second rule's URI attribute if their access does not match rule 2, regardless whether it would match rule 1 or not. Rule

2 then allows admin access to `http://hostname.net/app/auth/admin`, the implicit DENY making this the exclusive access right for admin.

- *Regular expression matching:* A completely different approach would be to give admin more control over what exactly the rule should match. Rather than merely deciding that the URI is to be interpreted as a prefix, we could use the requested resource's URI as a string that we try to match against a URI regular expression stored in the HBAC rule. If the regular expression matches, the rule matches. Another benefit is that we do not need rule ordering or DENY rules and we do not need to order rules by anything at all, not even prefix length.

  This approach is strictly more powerful than any of the above mentioned – when using Perl compatible regular expressions [26], we can still use prefix matching (using `.*` at the end), we can easily specify exceptions (using lookarounds – see the next example using negative lookahead), and we can even use more advanced matching not based on length or hierarchical URI structure (for example, we can grant access to pages of all users starting with `example-user-`, followed by a number). The goals specified for our example could then be easily achieved by (using PCRE):

  ```
  ALLOW any URI ^http://hostname.net/app/auth/(?!admin).*
  ALLOW admin URI ^http://hostname.net/app/auth/admin.*
  ```

  We use negative lookahead in rule 1 to except the case where `http://hostname.net/app/auth/` is immediately followed by `admin`. We also use `.*` to match the URI as a prefix. The second rule then allows admin's access to every resource starting with `http://hostname.net/app/auth/admin`.

  PCRE library is already used in SSSD so this approach would not add any dependency.

While the regular expression is more powerful, it is also less user-friendly:

- Admin needs to write a correct regular expression. It might not be easy in some cases and is error-prone.
- When adding a rule for a subset of URIs of another rule, it is necessary not only to add one rule, but also to change another - the rule matching the superset of rule being added, making an exception from that rule. This is another opportunity to make a mistake and forgetting this step would mean granting access to more users than should be granted access. This approach therefore leads to worse maintainability. For example, when we have rules:

  ```
  ALLOW any URI ^http://hostname.net/app/auth/(?!admin).*
  ALLOW admin URI ^http://hostname.net/app/auth/admin.*
  ```

  ... and we want to add a rule granting access to `http://hostname.net/app/auth/privileged-user` to privileged-user, we must add a rule allowing privileged-user to access that URI and also change rule 1 so that contains an exception (in our case, a negative lookahead) for that URI.

```
ALLOW any URI ^http://hostname.net/app/auth/(?!admin)(?!privileged-user).*
ALLOW admin URI ^http://hostname.net/app/auth/admin.*
ALLOW privileged-user URI ^http://hostname.net/app/auth/privileged-user.*
```

- Also after consulting with upstream, this seems like less-favorable solution than longest-prefix matching.

Therefore, we choose longest-prefix matching as an approach to interpret and match URIs in HBAC rules. It is more user-friendly and less error-prone. Also, while being strictly less powerful than the regular expression approach, it is not missing any important capability, especially with mainly web applications in mind – we usually only need to decide based on strict equality in each hierarchy level of URI.

7.1.2 Parts of URI to consider

As we described earlier, there are multiple parts of URI and some of them need to be matched in a way other than the others. The above-described longest prefix matching makes sense for the hier-part, query and fragment parts of URI (further referred to as path). It is considered case-sensitive by the standard, as also shown earlier. However, the scheme, host and port parts of URI should be matched differently in two ways:

- Comparison should be case-insensitive, as defined by standard, rather than case-sensitive rest of URI
- We do *not* want longest-prefix matching. It makes sense in a hierarchically structured path part of URI, but not for scheme, host and port. For example, `http://host.net` is completely different to `http://host.net.com` and these two should never be matched.

Therefore, we want to add attributes for two parts of URI, one with case-insensitive strict equality (scheme, host and port – `schemeAndHost`) and one with case-sensitive prefix matching (rest of the URI – `URI`). Also, when the `schemeAndHost` attribute is empty, we match it to any `schemeAndHost` in request – empty value serves as "any" because often, it is not desirable to only limit a rule to one `schemeAndHost`.

7.1.3 Relationship to other attributes

There are multiple attributes of the HBAC rule by which we match the rule to a request. So far, there are user, user group, host, host group, service, and service group categories. There are also name and description attributes which are not used for matching, they only serve for rule identification. After adding URI attributes, we want a rule to match if and only if it would match based on the old attributes *and* it matches in terms of both newly added attributes, both scheme, host and port attribute and rest of URI attribute, in the way we described they should be matched.

This means the new attributes do not affect HBAC rule evaluation in any way other than possibly making a rule that would otherwise match not to match, and the new attributes can be matched only after other attributes are matched if the only result expected is whether the sole rule matches or not. There is no dependency between these attributes at all.

7.1.4 Form and interpretation of URI attribute

As a result of discussion of mentioned problems, we conclude that:

- We will add two new attributes to the HBAC rule.
- One of them will be scheme, host and port part of the URI – it will be matched as case-insensitive strict equality.
- Another one will be the rest of URI. It will use property of URIs in many services – the fact that they are hierarchical. Only the rule with longest matching prefix will match as a whole rule.
- We only take in consideration the HBAC rules that would match if there were not for the new attributes.

7.1.5 Backwards compatibility

When using the old versions of the application, there is a problem: The old applications are not aware of URI-based authentication and that can not be changed by now, however, the new attributes can make the set of matching rules a subset of rules matching without them (while the rule not matching without the new attributes can never match with them). This makes the old application's behavior problematic: when they evaluate the rule, they will ignore the new attributes and can possibly match rules that should never be matched for the particular URI.

The same applies to the old versions of SSSD because while the PAM request contains the new attributes, SSSD does not know it should decide based on them and the situation is effectively the same as with the URI-based HBAC-unaware application.

When the PAM request does not contain the new attributes or there is an old version of SSSD used, there are two possible interpretations:

"Do not care" interpretation

We evaluate the rules as if the application did not care about the attributes value:

- If a request does not contain some of the new attributes, we understand that the client application either does not support URI-based HBAC or does not care about these arguments.

  If the application does not care about the new attributes, the result is simple – we do not try to match the attributes and only match the other ones. If they match, the rule matches. This also applies if just one of the new attributes is set – we ignore that one and only use another.

  If an application does not understand URI-based HBAC at all, we can only provide answer whether the triplet (user,service,hostname) matches. This is the way rules are evaluated now. If the rule matches based on these attributes, we consider it a matching rule. This notion should not cause any problem because the application unaware of the URI-based HBAC could only decide based on this triplet and the answer does not change with adding URI-based HBAC. *It is, however, important that the client application that performs URI-based HBAC does indeed include schemeAndHost and URI PAM parameters* lest the result can be allowing more access than should be allowed because SSSD will expect that the client application does not want URI and schemeAndHost to be considered during evaluation.

- If the HBAC rule does not contain some of the new attributes because SSSD is running against some older version of FreeIPA which does not support URI-based HBAC, we do not care about that attribute. If the rule would match with other attributes, it still matches, if it would not match, it still does not match. *It is important that the admin is aware that the client application is running against some older version of FreeIPA.* That is, however, not a problem because if they do not set rules for URI (which they do not as it is not possible in older FreeIPA versions), they naturally can not expect URI-based HBAC.

  If schemeAndHost attribute is empty, anything matches because we expect the attribute not set means it is not important, rather than scheme and host being empty. If the rest of URI is empty, it matches everything because an ampty string is prefix of every string.

- The same applies if both the request and the rule misses some or all of the new attributes or some of the attributes is empty.

When using this interpretation, however, there can be more rules matched than specified in FreeIPA: any rule that would match in URI-unaware HBAC matches here, too. When the cause of missing attributes is an application that is aware of URI-based HBAC and uses the lack of attribute to indicate – correlating to the specification – that it does not care about the attribute's value, this interpretation is fine. The problem is when there is an old version of the application or SSSD, together with a new version of FreeIPA, involved. While there is a rule in FreeIPA present that only allows access to certain (user,host,service,URI), the SSSD evaluates HBAC rules only by (user,host,service), potentially matching a rule that is not intended by admin to match. Consider, for example, the rule:

```
ALLOW anyone HOST any SERVICE any URI http://host/login
```

With this rule and the old version of SSSD or application, the PAM request would not contain any of the new attributes and consistently with URI-unaware HBAC, the rule would match while the actual URI accessed might have been `http://host/admin`. This behavior is not what FreeIPA admin obviously intended by that rule.

"Empty value" interpretation

Rather than expecting the application not to care about the unset attributes, we should consider the possibility of the application being simply unaware of URI-based HBAC and administrator willing to set fine-grained URI-based HBAC with some applications running their old versions. In that case, we should design the changes in a way ensuring lack of attributes in the authorization request will not lead to matching more rules that intended by administrator.

We will do this by having actually two types of rules in FreeIPA: those for URI-aware HBAC and those for URI-unaware HBAC, while the latter will keep the same form as they had in the previous versions and the former will be made in such a way that they will be ignored by older versions of SSSD. The new version of SSSD must be able to handle both these types.

This makes URI-based HBAC behavior backwards compatible with behavior of URI-based HBAC unaware older versions (as we can not change the old applications' behavior) while preventing situations when the rules matched by the new infrastructure are a proper subset of those matched by the old infrastructure for the same resource access.

This approach is also very consistent with the meaning of empty attribute if we did not do any special interpretation (i.e., we did not care about backwards compatibility at all) – in case of empty attribute, we would only match rules with their URI attribute being a prefix of the empty string, that means only rules with URI attribute empty.

7.2 FreeIPA side changes

On FreeIPA side, we need to:

- Allow storing of the two additional attributes – schemeAndHost (scheme, host and port part of URI) and URI (the rest of URI). These two attributes should be orthogonal – they do not depend on each other and one, both or neither of them can be set. The data types should allow correct handling (namely comparison) of each of them to support LDAP search in them.

  Because of backwards compatibility (as discussed in section 7.1.5), there will be actually two types of HBAC rules: those for URI-unaware HBAC (the same as until now) and those for URI-aware HBAC. This HBAC rule duality means adding another entity and changing its type according to value of the new attributes.

- Make the attributes accessible via LDAP.

- Allow manipulating attributes:

  - Setting schemeAndHost attribute.
  - Setting URI attribute.
  - Modifying schemeAndHost attribute.
  - Modifying URI attribute.
  - Removing (setting to empty) schemeAndHost attribute.
  - Removing (setting to empty) URI attribute.

  Using:

  - API
  - WebUI

7.3 Communication between FreeIPA and SSSD

LDAP protocol is used for communication between FreeIPA and SSSD because it is just getting some data from FreeIPA's LDAP. Nothing is being evaluated or changed on FreeIPA side. SSSD's IPA provider uses SDAP [32] provider which uses LDAP provider to get the data.

We only need to allow SSSD's access to the attributes (set correct acces rights for these attributes) in FreeIPA and do changes on SSSD's side so that it requests the new attributes over LDAP.

7.4 SSSD side changes

We need LDAP to get the HBAC rule information from FreeIPA, get the authorization request with necessary data from some client application, and evaluate the request based on HBAC rule information available. We also want the SSSD with this change implemented to be backwards compatible.

HBAC rules are received through LDAP so we must make sure we request the new attributes as well as the old ones. For backwards compatibility, we must correctly handle the response not containing these new attributes. For the same reason, we also must be able to decide based on both the rules for URI-unaware HBAC and for URI-aware HBAC.

Request from client should be received over PAM responder, so we must make sure it understands the new request parameters containing schmemeAndHost accessed and URI of the requested resource. We must make sure this PAM responder forwards the request with the new attributes correctly to be evaluated. For backwards compatibility, we must not require the new attributes to be set in PAM request and correctly interpret the request while matching rules when the new attributes are indeed not present. This means we must understand both forms of rules and when the request contains some of the new attributes, act on all of them, otherwise only use the rules for URI-unaware HBAC.

7.5 Communication between SSSD and Apache module

The client application will communicate with SSSD using PAM. The PAM service must be configured to use `pam_sss` library. This library must be changed so that it supports URI-based HBAC. `pam_sss` is part of SSSD project.

The new request attributes will be sent as PAM environment variables (schemeAndHost and URI) because there are no suitable PAM items for this purpose [25]. The client application must therefore set the right values to these PAM environment variables and `pam_sss` must read them and send them further to SSSD's PAM responder. These environment variables are not the standard PAM fields – they will be ignored by applications unaware of URI-based HBAC.

7.6 Client application

URI-based HBAC functionality is not limited to one particular application. However, we have mainly web applications in mind when proposing it. For this reason, we will use the Apache HTTP Server as an example for this thesis. We will write an Apache module that, after being called from Apache HTTP Server's authorization hook, uses `pam_sss` (using PAM interface) to verify user's access rights to the requested resource.

Because of how we designed the feature, the application does not need to evaluate anything, it merely asks for authorization through a standardized PAM interface and receives the authorized/unauthorized response. The only difference to current usage of HBAC is that it also needs to set schemeAndHost and URI PAM environment variables if we want them to be taken into consideration.

Chapter 8

Implementation

In this part, we will describe some technical details and difficulties of implementing the concept described in the previous chapter.

While also the version with matching using regular expressions has been implemented, we focus mainly on the longest prefix matching version as it is easier to maintain, less error-prone and better acknowledged by upstream.

8.1 FreeIPA

On FreeIPA side, we must do changes to LDAP schema to add data types for the new attributes and make them part of HBAC rule. We then need to change API and WebUI so that it makes these attributes available and modifiable.

8.1.1 LDAP

FreeIPA's LDAP schema is defined in `install/share/60basev2.ldif`. It is an LDIF file [1]. LDIF files (LDAP Data interchange Files) are plaintext files intended for exporting and importing LDAP directory data. The file is used on directory server initialization. We must change the `ipaHBACRule` object to contain additional `URI` and `schemeAndHost` attributes and define these attributes to have suitable data types.

To add attributes, we use the `attributeTypes` statement. For each of the attributes, we must specify:

- `OID` - Object Identifier number
- `NAME` - Name of the attribute
- `DESC` - Description of the attribute
- `EQUALITY` - How two attributes of this type are checked for equality
- `ORDERING` - How attributes of this type are ordered
- `SUBSTR` - How an attribute of this type is checked for being substring of another attribute of the same type
- `SYNTAX` - How the attribute is represented
- `X-ORIGIN` - Defining where the attribute was originally defined

Attribute	URI	Scheme, host and port
OID	2.16.840.1.113730.3.8.11.73	2.16.840.1.113730.3.8.11.74
NAME	uri	schemeAndHost
DESC	Path part of URI	Scheme, host and port part of URI
EQUALITY	caseExactMatch	caseIgnoreMatch
ORDERING	caseExactOrderingMatch	caseIgnoreOrderingMatch
SUBSTR	caseExactSubstringsMatch	caseIgnoreSubstringsMatch
SYNTAX	1.3.6.1.4.1.1466.115.121.1.15 (UTF-8 string)	1.3.6.1.4.1.1466.115.121.1.15 (UTF-8 string)
X-ORIGIN	IPA v3	IPA v3

These parameters define how the attribute should be represented, what is its semantics and lets us use LDAP's functions to process, filter or order the attributes in an LDAP request.

We want to add an URI attribute with UTF-8 string syntax because URI is expressed as such a string [43]. Contents of the attribute should be handled in case-sensitive manner, thus the caseExact* definitions for EQUALITY, ORDERING and SUBSTR. We will use OID from IPAv3 space (2.16.840.1.113730.3.8.11.*).

We also want to add a schemeAndHost attribute using the same syntax but using the case-insensitive options for EQUALITY, ORDERING and SUBSTR because, as we described in chapter 7.1, it should be understood as a case-insensitive string. Both attribute types' parameters are shown in table 8.1.1.

To make the attributes part of the ipaHBACRule object, we use the objectClasses statement. We will use the following parameters [21]:

- OID - Object Identifier number
- NAME - Name of the object class
- SUP - Superior object classes
- TYPE - Type of object class
- MUST - Required attributes
- MAY - Allowed attributes
- X-ORIGIN - Defining where the object class was originally defined

To add the two attributes to the existing ipaHBACRule class, we can keep most of the parameters intact. We only add the URI and schemeAndHost objects to the allowed attributes list, i.e. the MAY part, because these two attributes are not supposed to be mandatory (missing attributes shall be handled as described in section 7.4).

Because of HBAC rule duality as described in section 7.1.5, we actually need one more object class. The ipaHBACRule class will represent old URI-unaware HBAC rules while the new class ipaHBACRuleURI will represent the new rules only available to SSSD versions that are URI-aware, i.e. the first version of SSSD capable of accessing these rules is the very version we are designing in this work.

Creating another class will work because SSSD gets the HBAC rules based on their class – ipaHBACRule. The older versions will not ask for the rules of class ipaHBACRuleURI and therefore will not see them and will not have opportunity to falsely match them to the request while they should not be matched.

8.1.2 API, WebUI

We need to add ways to add, delete and modify `URI` and `schemeAndHost` attributes using both API and WebUI. FreeIPA's API is implemented in `ipalib` which is a Python module. We extend `hbacrule` class (a subclass of LDAPObject) by these attributes and new methods. API is generated based on this object, thus it has commands `ipa hbacrule-add`, `ipa hbacrule-del`, `ipa hbacrule-mod`, `ipa hbacrule-find` and `ipa hbacrule-show` with proper parameters (exactly described in command's help).

With adding the new class for the URI-aware HBAC rules, we also need to make sure the API can correctly work with it, especially:

- On modifying the rule, change its `objectClass` according to values of URI and schemeAndHost attributes: set it to `ipaHBACRule` when they are both empty and to `ipaHBACRuleURI` when some of them is not.
- When searching for or listing HBAC rules, search or list both the new and old class rules.
- Correctly modify, delete etc. rules of both types

We will base these modifications on the old class `hbacrule` and will not add any other class, keeping the API unchanged (except for adding two new attributes) and making the HBAC rule duality transparent – the user will never know there are two types of HBAC rules in the first place.

FreeIPA's WebUI uses Freeipa's API to manage LDAP's content. It is based on Javascript and uses JSON-RPC to communicate with API [38][19]. User interface is modular and consists of *facets* built of *widgets*. To add ways for user to manipulate new attributes using WebUI, we must therefore create new widgets for manupulating the attributes and use the widgets in the HBAC rule's facet. Because WebUI uses API, we do not need to modify add/mod/delete/list operations specifically for WebUI as we have already done this in API.

8.2 SSSD

We need to make SSSD capable of getting HBAC rules from FreeIPA, receiving authorization request from the client application and evaluate the request based on data provided by the client applicaiton and HBAC rules.

8.2.1 Getting rules

`URI` and `schemeAndHost` are attributes of the ipaHBACRule object in FreeIPA's LDAP. We need to make these data available to HBAC rule evaluator. HBAC rules are evaluated in SSSD's IPA provider plugin.

The IPA provider plugin, upon receiving authorization request, gets the HBAC rules from SSSD's cache. The cache is updated during this request so it is up to date. To receive an attribute of ipaHBACRule object, the IPA provider plugin must ask for it excplicitly so we need to add `URI` and `schemeAndHost` to the requested attributes. We need to ask for data both of class `ipaHBACRule` and of class `ipaHBACRuleURI` so we change the filter accordingly. When the data in cache is not present or not up to date, SDAP provider plugin [32] is used to get the data. SDAP provider plugin is itself a wrapper over LDAP provider plugin which gets the data directly from FreeIPA's LDAP using standard LDAP.

8.2.2 PAM responder

PAM responder is a part responsible for receiving PAM requests from the application (`pam_sss` library), processing them and calling proper routines to evaluate the request. It listens on `AF_UNIX` named pipe `${localstatedir}/lib/sss/pipes/pam` [15] where `${localstatedir}` is prefix, typically `/var`.

We extend the application protocol used between `pam_sss` and PAM responder to be able to carry the two additional attributes. We make PAM responder understand `URI` and `schemeAndHost` so that it saves them upon receiving rather than throwing protocol error. We also change the interface between PAM and IPA provider plugin so that it is called with additional data.

`URI` and `schemeAndHost` attributes should be set to empty string when they are missing in the request, ensuring backwards compatibility – older `pam_sss` versions do not send the attributes but PAM responder makes the interface with rest of SSSD consistent for both old versions and new versions, both sending and not sending the attribute data (even requests from the new versions of `pam_sss` need not contain the new attributes).

Upon receiving the request, the request is forwarded to the correct provider plugin – IPA provider plugin in this case. The communication between PAM responder and IPA provider plugin is done using S-BUS (which is a subset of D-Bus protocol using `libdbus-1` library and implementing a subset of its features) listening on UNIX socket `/var/lib/sss/pipes/private/sbus-dp_${domain_name}` [15].

8.2.3 Evaluating rules

When we call ipa provider with requested values and it downloads rules from FreeIPA, we have all the information necessary to decide whether we should authorize the access or not. We compare each part – schemeAndHost and URI – separately.

We compare rule by rule, taking into account rules both of class `ipaHBACRule` and of class `ipaHBACRuleURI`. We try to find the longest-prefix match of URI therefore, while cycling through rules, we remember the result of every comparison of a rule URI attribute of which is longer prefix of the requested URI than the previous remembered rule. If the prefix length is equally as long as of the previous remembered result, and either the previous remembered result or the result of last comparison is ALLOW, then we remember ALLOW as the new result. This is because the same length combined with case-sensitive prefix matching means that URI attributes in those rules are equal and the semantics of HBAC rules is that when at least one of the rules allows access, the access is to be allowed.

The matching is done in manner justified in the previous chapter:

- We only attempt to match schemeAndHost if the rule matches in terms of host and service.
- When comparing schemeAndHost, we do so using SSSD's function `sss_utf8_case_eq`. This function does exact match comparison except it is case-insensitive. We check whether the currently compared rule's schemeAndHost attribute matches the schemeAndHost specified in request, according to this function.
- We only attempt to match URI if the rule matches in terms of schemeAndHost.

```
$result \leftarrow DENY$;
$longestPrefixLength \leftarrow 0$;
forall $rules$ do
    $intermediateResult, prefixLength \leftarrow evaluateRule(rule)$;
    if $intermediateResult = UNMATCHED$ then
        if $prefixLength > longestPrefixLength$ then
            $longestPrefixLength \leftarrow prefixLength$;
            $result \leftarrow DENY$;
        end
    else
        if $prefixLength \geq longestPrefixLength$ then
            $longestPrefixLength \leftarrow prefixLength$;
            $result \leftarrow ALLOW$;
        end
    end
end
return $result$
```

Figure 8.1: Algorithm used for rule evaluation

- When comparing URI, we check whether the currently compared rule's URI attribute is a prefix of the URI specified in request. As empty string is a prefix of every string, the rule with empty URI attribute matches any request regarding its URI.

As the result of rule evaluation, we return the last remembered value. The algorithm can be also described by algorithm 8.1.

8.3 pam_sss

`pam_sss` is a part of the SSSD project. We need to change it so that application using PAM with service that has defined `pam_sss` as a provider is able to send request containing the new parameters to the running `SSSD` daemon. As we described earlier, the application uses standard `PAM` request for this task, but also specifies non-standard `PAM` environment variables understood by applications aware of URI-based HBAC. We therefore need to make `pam_sss` understand these additional arguments: `URI` and `schemeAndHost`.

When `pam_sss` receives a request, it uses `pam_getenv` call to get values of these attributes. They are set to empty if they are not set by the aplication – for backwards compatibility, they are not required.

Upon receiving all the information necessary, `pam_sss` uses `AF_UNIX` socket to communicate with FreeIPA's PAM responder and request authorization, using the parameters received from the application. We therefore make sure we extend the communication protocol by these two new arguments and actually send them.

8.4 Apache modules

To have a working example of application using URI-based HBAC, we have made a module for Apache HTTP Server which will enhance its authorization capabilities by URI-based

HBAC. The module implements authz_provider interface meant for authorization-related Apache modules.

8.4.1 mod_hbacauthz_pam

As a solution meant solely for URI-based HBAC, we have made a new module which presumes the authentication has already taken place and the authenticated user's name is saved in its r->user variable where r is a request_rec type variable passed automatically to the called Apache modules. To keep the module as simple as possible, we only make it compatible with Apache HTTP Server 2.4. That is because Apache API changed for AAA in this version. It should not pose a problem because Apache HTTP Server 2.4 is a version from beginning of year 2012 [5].

The application's (module's) only tasks are to gather information necessary to send a PAM request (user, host, service, scheme, port, URI), send this request with properly set PAM items *and PAM variables with the new attributes*, and receive the binary authorized/unauthorized answer. The sources for information we need are:

- *User*: r->user
- *Host*: There are two different "Host" parameters:
  - The Host corresponding to Host attribute of HBAC rule. Its value is implicitly decided by the hostname of machine asking for PAM authorization – which is, from the point of SSSD, hostname of the very machine SSSD is running on. This item does not therefore need to be sent as a part of the PAM request.
  - The Host that will be part of the schemeAndHost attribute. It could be the hostname that client (the web browser requesting the page) actually used in URI or canonical hostname of the virtual host handling the HTTP request. The Host header is, however, not mandatory in HTTP/1.0 [20]. Furthermore, this header can be spoofed and should not be relied upon for security applications. For this reason, we use the second option. We get this value using ap_get_server_name which returns the canonical name of the virtual host handling the HTTP request. Often, these two notions of Host result in the same value, sometimes, however, they do not, e.g. when there are multiple virtual HTTP servers running on a single machine (which has multiple DNS records pointing to it).
- *Service*: Specified in the conf/http.conf file or in some of the .conf files in conf.d/ directory in Apache's settings directory (typically /etc/httpd/). It is available in the Apache modules using ap_getword_conf(r->pool, &require_args).
- *Scheme*: ap_http_scheme(r)
- *Port*: ap_get_server_port(r)
- *URI*: r->uri which already contains only the path part (it is stripped of scheme, host and port)

This information is then sent using PAM interface – either its standard items or PAM environment variables (the pam_putenv call) in case of URI and schemeAndHost attributes. Upon setting all the data necessary, pam_acct_mgmt call is executed which returns either AUTHORIZED or NOT_AUTHORIZED and this is also returned as the result of module's call.

8.4.2 mod_authnz_pam

We can also enhance the existing module `mod_authnz_pam` to send the new attributes and therefore make its built-in authorization part into URI-aware HBAC. We will do this very similarly to `mod_hbacauthz_pam`.

8.5 Differences for PCRE-based matching strategy

We decided *not* to use the regular expression approach to interpretation and comparison of URI paths. However, changing this decision and making URI-based HBAC based on PCRE would be very easy.

The only part that would need to be changed is the SSSD evaluation part. As PCRE library is already used in SSSD, this would not add any dependences. The evaluating strategy would be the same, except that we would not need to find the longest-prefix match and we could ALLOW access whenever we find the first rule that matches. Matching would be very straight-forward, instead of checking whether URI path attribute of the rule is a prefix of the requested URI path, we would check whether it is a regular expression describing a set of strings such that the requested URI path is part of this set.

Chapter 9

Testing

After implementing the whole URI-based HBAC infrastructure, we need to make sure that:

- The solution works – we can indeed control access based on URI
- It works reasonably fast
- Other requirements specified in chapter 4 are satisfied

9.1 Functionality testing

We can divide functional requirements to two parts. Whether URI-based HBAC, once properly set, works, and whether it is possible to manage the HBAC rules using proper tools.

9.1.1 Tools

By inspecting help of the `ipa` command on FreeIPA server, we can conclude we are able to add, change and delete both URI and schemeAndHost attributes of a HBAC rule. It can be done by commands `ipa hbacrule-add`, `ipa hbacrule-del` and `ipa hbacrule-mod` with proper parameters (exactly described in command's help).

In the same way, by inspecting the FreeIPA WebUI, we see we can do the very same operations using webUI, which itself is using IPA API. It is in the *Policy / Host Based Access Control* section.

We can as well see that the new arguments are readable (and *not* modifiable) using LDAP tools against FreeIPA's LDAP.

9.1.2 URI based HBAC

Given the rules are properly set, we want to test that URI-based HBAC actually works. Unit tests are part of the SSSD and FreeIPA projects and are easily understandable from the source code. We have, however, manually tested the basic cases:

- There is a rule allowing access to certain user, host, service and URI and there is no other rule with its URI attribute being a longer prefix of the requested resource's URI than that rule. We expect access to be allowed. For requested resource's URI /application/login, the used rules were (host and service and schemeAndHost attribute omitted and expected to match):

```
ALLOW anyone URI /application
ALLOW anyone URI /whatever
```

- There is a rule allowing access to certain user, host, service and URI but there is another rule with its URI attribute being a longer prefix of the requested resource's URI than that rule *and* not allowing that user to access *and* there is no other rule that would in the similar manner allow that user's access again. We expect access to be denied for the user. We also expect access to be allowed to the user specified in that second rule. Example of such rules (for user *not* being `admin`):

  ```
  ALLOW anyone URI /application
  ALLOW admin URI /application/login
  ```

- There is a rule with matching schemeAndHost but not matching URI. There is no other matching rule. We expect access to be denied.
- There is a rule with matching URI but not matching schemeAndHost. There is no other matching rule. We expect access to be denied.
- There is a rule matching without URI-wise HBAC and without any of the new arguments specified. We expect access to be allowed based on such a rule.

9.2 Performance

To test whether the changes did not slow down evaluation too much, we test authorization 20 times against FreeIPA with 256 rules specified, first using the old URI-based HBAC-unaware software and then using software modified for URI-based HBAC together with `mod_hbacauthz_pam` module, on the same machine and OS. We make an average of time of those 20 tries for each version and compare the averages.

For standard HBAC unaware of URI, the average time to access (or get access denied) the `mod_authnz_pam` secured page was *165 ms*. For URI-based HBAC, the average time to access the `mod_hbacauthz_pam` secured page was *166 ms*. We see the difference is insignificant and it is less than measuring error. From that, we can conclude that performance is not largely affected by URI-based HBAC.

9.3 Example of use

We can furthermore demonstrate usability of the example on securing a well-known real-world application. For its widespread use, we have chosen to secure a WordPress instance with URI-based HBAC. We will use clean WordPress installation (with address `http://$(hostname)/wordpress`), add a plugin ensuring that WordPress respects the `REMOTE_USER` server variable and set up Kerberos authentication for the site with Kerberos server being the one running as part of FreeIPA. Such a WordPress instance has three important parts from security point of view:

- Public part (`/wordpress`) which anyone can access, even unauthenticated user
- Maintenance part `/wordpress/wp-admin` which any authenticated user can access to add and edit posts etc., except for parts which are meant for admin only

- Admin part `/wordpress/wp-admin/X` for X being some of admin-specific parts of maintenance part – only admin (in our example, user `wpadmin`) can access here

After installing the WordPress instance, setting up authentication and authorization and setting proper rules in FreeIPA, we will check for unauthenticated user, non-admin and wpadmin whether they can access exactly what they should be able to access. The rules are the following (we can – but do not have to – also setup schemeAndHost values to `http://$(hostname):80` if we want to set these rules for one specific HTTP host, otherwise they match access to any HTTP host with the same path):

```
ALLOW <anyone> SERVICE wordpress URI /wordpress/wp-login.php
ALLOW <anyone> SERVICE wordpress URI /wordpress/wp-admin/
# FOR X in themes,customize,widgets,nav-menus,theme-editor,plugins,
# plugin-install,plugin-editor,users,user-new,options-general,
# options-writing,options-reading,options-discussion,options-media,
# options-permalink
ALLOW admin SERVICE wordpress URI /wordpress/wp-admin/X
# ENDFOR
```

We can add these rules by using the following API calls:

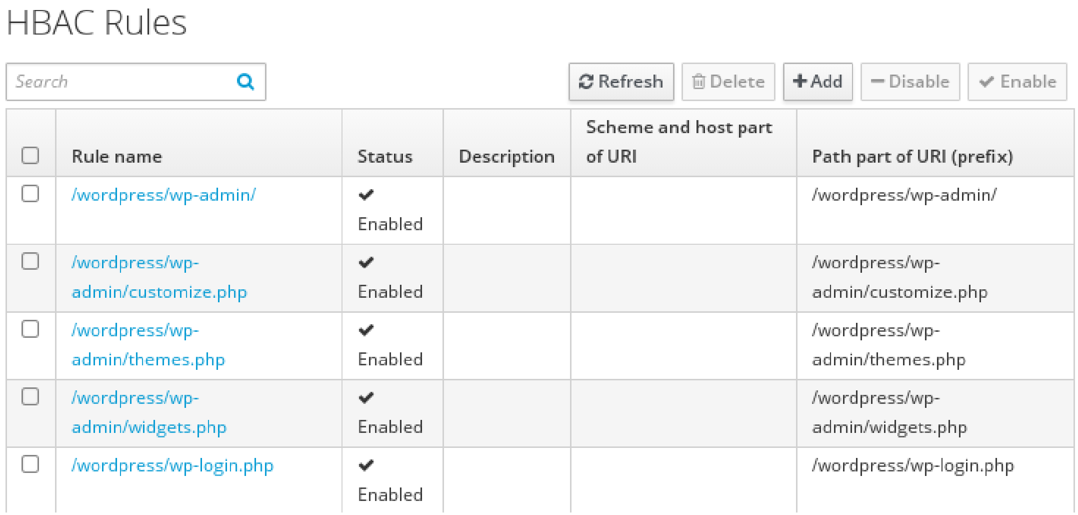
```
# add the HBAC service
ipa hbacsvc-add wordpress

# add the rule for wp-login.php, any authenticated user can access
# on any machine
ipa hbacrule-add /wordpress/wp-login.php --url='/wordpress/wp-login.php'
 --usercat=all --hostcat=all
# the rule is only valid for wordpress HBAC service
ipa hbacrule-add-service /wordpress/wp-login.php --hbacsvcs=wordpress

# add the rule for wp-admin, any authenticated user can access
# on any machine
ipa hbacrule-add /wordpress/wp-admin/ --url='/wordpress/wp-admin/'
 --usercat=all --hostcat=all
# the rule is only valid for wordpress HBAC service
ipa hbacrule-add-service /wordpress/wp-admin/ --hbacsvcs=wordpress

# if wp-admin/ is followed by one of these, use stricter rules
for admincat in {themes,customize,widgets,nav-menus,theme-editor,plugins,
plugin-install,plugin-editor,users,user-new,options-general,
options-writing,options-reading,options-discussion,options-media,
options-permalink}.php; do

        # create a HBAC rule for admin-only parts
        ipa hbacrule-add /wordpress/wp-admin/$admincat
```



□	Rule name	Status	Description	Scheme and host part of URI	Path part of URI (prefix)
□	/wordpress/wp-admin/	✓ Enabled			/wordpress/wp-admin/
□	/wordpress/wp-admin/customize.php	✓ Enabled			/wordpress/wp-admin/customize.php
□	/wordpress/wp-admin/themes.php	✓ Enabled			/wordpress/wp-admin/themes.php
□	/wordpress/wp-admin/widgets.php	✓ Enabled			/wordpress/wp-admin/widgets.php
□	/wordpress/wp-login.php	✓ Enabled			/wordpress/wp-login.php

Figure 9.1: HBAC rule list in WebUI

```
         --url="/wordpress/wp-admin/$admincat" --hostcat=all
        # the rules are only valid for wordpress HBAC service
        ipa hbacrule-add-service /wordpress/wp-admin/$admincat
         --hbacsvcs=wordpress
        # the rules only allow wpadmin user, no one else,
        # even if he is authenticated
        ipa hbacrule-add-user /wordpress/wp-admin/$admincat --user=wpadmin

done
```

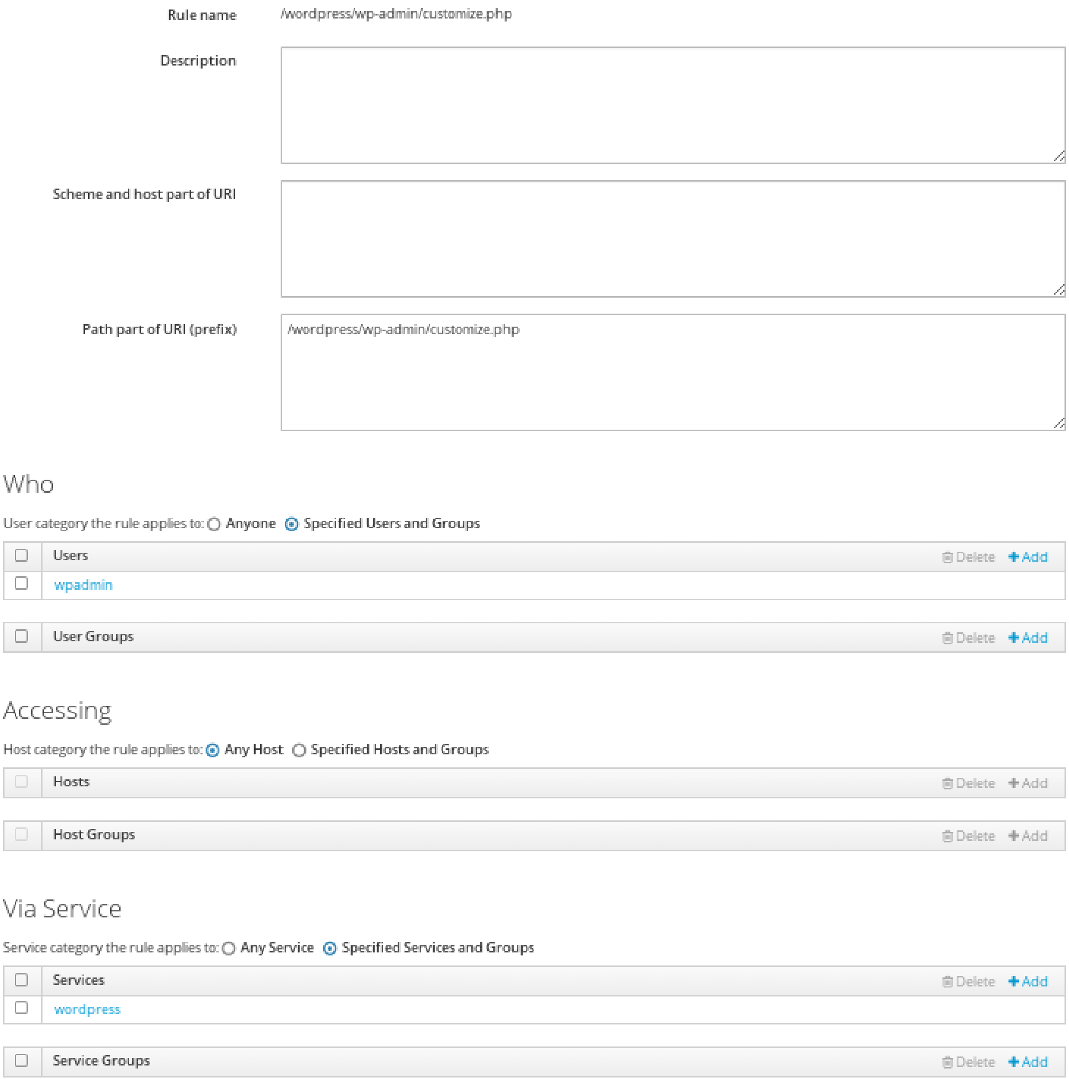
The resulting list of rules in WebUI is shown in figure 9.1 and rule detail page is shown in figure 9.2.

We will set up Apache HTTP Server to use Kerberos authentication on `/wordpress/wp-login.php` and `/wordpress/wp-admin/*`. We will use WordPress's `http-authentication` plugin to understand the `REMOTE_USER` variable and log in the user based on their Kerberos ticket. We will setup the Apache HTTP Server to use `mod_hbacauthz_pam` module for authorization. We do not need to setup anything for public part because users are not required to be authenticated nor authorized there and anyone can access it.

9.4 How to setup WordPress with Kerberos and URI-based HBAC

In this section, we will describe the steps to setup the WordPress instance in manner described above. To make it easier, a few scripts are included to automate things such as FreeIPA server installation or securing WordPress with proper HBAC rules. These scripts are:

- `make.sh` – compiles FreeIPA if run with first argument `ipa`, SSSD if run with `sssd` or both if run with `both`



Rule name: /wordpress/wp-admin/customize.php

Description:

Scheme and host part of URI:

Path part of URI (prefix): /wordpress/wp-admin/customize.php

Who

User category the rule applies to: ○ Anyone ● Specified Users and Groups

□	Users
□	wpadmin

□	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
□	wordpress

□	Service Groups

Figure 9.2: HBAC rule detail in WebUI

- install_server.sh – based on information set in scripts.conf, installs the FreeIPA server
- install_client.sh – based on information set in scripts.conf, installs the FreeIPA client (sets up SSSD, Kerberos etc.)
- install.sh – runs make.sh with first argument equal the script's first argument, install_server.sh and install_client.sh
- secure_wordpress.sh – when run on FreeIPA server, adds the rules securing a WordPress instance with PAM service name equal to the first argument and using the second argument as WordPress administrator's login (other authenticated users will have access rights corresponding to Author privileges in WordPress)

The scripts.conf file is used to set some necessary information to perform installation, mainly FreeIPA server's hostname and the client's hostname. These tests should be run from the freeipa-scripts directory. These tests have not been tested extensively but should work fine on Fedora 23. Using these scripts, we can make the described setup by executing these steps:

1. Install modified versions of FreeIPA and SSSD (you can use the install.sh script)
2. Setup a WordPress instance on client's Apache HTTP Server in /wordpress
3. Add the users you want to use to both WordpPess and FreeIPA (do not forget to include domain in WordPress)
4. Setup http-authentication plugin in the WordPress instance [13]
5. Install mod_hbacauthz_pam plugin in the Apache HTTP Server's module directory if not done by the install script before, and restart Apache HTTP Server
6. Setup Apache HTTP Server to use mod_hbacauthz_pam in the correct Apache Locations (/wordpress/wp-login.php and /wordpress/wp-admin/*) by using require pam-account <pam_service_name> directive
7. Setup Kerberos authentication in the same Locations
8. Configure PAM to use pam_sss for authorization for chosen PAM service name in /etc/pam.d/<pam_service_name>
9. Run secure_wordpress.sh script on the FreeIPA server to add correct HBAC rules

Now, with Kerberos ticket other than wpadmin's, user can not access admin-only parts of WordPress:

```
[root@host ~]# kinit user42
[root@host ~]# curl -i -u : --negotiate http://$(hostname)/wordpress/\
             # wp-admin/customize.php
HTTP/1.1 401 Unauthorized
Date: Fri, 20 May 2016 13:15:01 GMT
Server: Apache/2.4.18 (Fedora) mod_auth_kerb/5.4
WWW-Authenticate: Negotiate
```

```
Content-Length: 63
Content-Type: text/html; charset=iso-8859-1

<html><body>Unauthorized.</body></html>
```

9.5 Results

With not much effort, we were able to secure a WordPress instance on webserver level, without needing any application-level access management. We can use FreeIPA's abilities of identity management, grouping services and hosts etc. for easier maintenance and to efficiently centrally manage access rights. We can see that URI-based HBAC can serve as a solution adding one more level of security to the system or can even be used standalone, while being able to determine not only based on the fact that the service is WordPress running on some host, but to provide fine-grained access control on level of exact address of the required resource.

Prefix matching makes it easy to use rules for large parts of the web, not needing to make a rule for every resource specifically, while longest-prefix matching allows for intuitive setting of exclusive access rights for some users, even if the part of web in consideration is a subpart of a larger part that, generally, allows more users' access.

To secure WordPress, just a few rules are necessary and their maintenance is made easy. Also, URI-based HBAC did not make system noticeably slower. Overally, the system seems to work in this case and can be expected to work in most scenarios.

Chapter 10

Conclusion

I created a set of tools together supporting identity management and access control based on URI of the requested resource. This allows for centralized management of access privileges, authorization based on request of providers of services and security on multiple levels. I use FreeIPA for management and storing of access rules with URI-related objects and their attributes. I use SSSD as an evaluating and caching service allowing for more efficient use and easier implementation of service-side part. I use PAM interface to further simplify the interface for the application, thus making it easier to implement URI-based access control in applications currently not supporting it. Using rule duality, I ensured backwards compatibility. I also created an Apache module demonstrating use of the tools and showed the configuration and usage example using a well-known web application, WordPress.

These tools together make a great improvement to current state of access-control in environments using FreeIPA identity management tool. Currently, the patches are being reviewed by upstreams. In conclusion, the solution seems to be useful and rigid enough and after being acepted by FreeIPA upstream, it can be used by wide audience.

10.1 Future work

Most of the future work should focus on further improving the tool based on the knowledge gained from usage in production environments. Also, before distributing the new version of FreeIPA containing the tool to enterprise environments (e.g., using it in the new version of Red Hat Enterprise Linux), more thorough user documentation should be written.

Bibliography

[1] A LDIF File Format. https://docs.oracle.com/cd/E10773_01/doc/oim.1014/e10531/ldif_appendix.htm. Online; Accessed: 2015-12-28.

[2] About - FreeIPA. https://www.freeipa.org/page/About. Online; Accessed: 2015-12-26.

[3] About Host-Based Access Control. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Linux_Domain_Identity_Authentication_and_Policy_Guide/configuring-host-access.html. Online; Accessed: 2015-12-27.

[4] Addressing and names in D-Bus. http://maemo.org/maemo_training_material/maemo4.x/html/maemo_Platform_Development_Chinook/Chapter_01_DBus_The_Message_Bus_System.html. Online; Accessed: 2015-12-30.

[5] [ANNOUNCEMENT] Apache HTTP Server 2.4.1 Released. http://marc.info/?l=apache-httpd-announce&m=132983471818384&w=2. Online; Accessed: 2016-04-28.

[6] Apache – HTTP server project. https://httpd.apache.org/. Online; Accessed: 2015-12-13.

[7] Apache module mod_authnz_pam. https://fedorahosted.org/webauthinfra/wiki/mod_authnz_pam. Online; Accessed: 2015-12-30.

[8] D-Bus Interface: Users and Groups. https://fedorahosted.org/sssd/wiki/DesignDocs/DBusUsersAndGroups. Online; Accessed: 2015-12-29.

[9] DBus responder design. https://fedorahosted.org/sssd/wiki/DesignDocs/DBusResponder. Online; Accessed: 2015-12-29.

[10] FreeIPA – identity | policy | audit. http://www.freeipa.org. Online; Accessed: 2015-11-15.

[11] FreeIPA Trac. https://fedorahosted.org/freeipa/. Online; Accessed: 2015-11-16.

[12] How the Kerberos Version 5 Authentication Protocol Works. https://technet.microsoft.com/en-us/library/cc772815%28v=ws.10%29.aspx. Online; Accessed: 2015-12-25.

[13] HTTP Authentication. https://wordpress.org/plugins/http-authentication/installation/. Online; Accessed: 2016-04-29.

[14] Identity Management Guide. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Identity_Management_Guide/ipa-linux-services.html. Online; Accessed: 2015-12-27.

[15] Inter-process communication between SSSD processes. https://fedorahosted.org/sssd/wiki/DesignDocs/IPC. Online; Accessed: 2016-04-10.

[16] Internet Engineering Task Force. https://www.ietf.org/. Online; Accessed: 2015-11-17.

[17] Introduction to Directory Services and Directory Server . https://docs.oracle.com/cd/E19396-01/817-7619/intro.html. Online; Accessed: 2015-12-27.

[18] IPAv3 Layout. http://www.freeipa.org/images/7/72/IPAv3-Layout.png. Online; Accessed: 2015-12-27.

[19] JSON-RPC. http://www.jsonrpc.org/. Online; Accessed: 2016-04-10.

[20] Key Differences between HTTP/1.0 and HTTP/1.1. http://www8.org/w8-papers/5c-protocols/key/key.html. Online; Accessed: 2016-04-29.

[21] LDIF Format for Adding Schema Elements. https://docs.oracle.com/cd/B14099_19/idmanage.1012/b15883/ldif_appendix003.htm. Online; Accessed: 2016-04-10.

[22] Linux Domain Identity, Authentication, and Policy Guide. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Linux_Domain_Identity_Authentication_and_Policy_Guide/ipa-linux-services.html. Online; Accessed: 2015-12-27.

[23] Package: freeipa-server. https://packages.debian.org/sid/net/freeipa-server. Online; Accessed: 2015-11-16.

[24] PAM Documentation. http://uw714doc.sco.com/en/SEC_pam/pamintro.html. Online; Accessed: 2015-12-30.

[25] pam_set_item - set and update PAM informations . http://linux.die.net/man/3/pam_set_item. Online; Accessed: 2016-04-09.

[26] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/. Online; Accessed: 2016-04-09.

[27] Proposal: drop DENY rules from HBAC. https://www.redhat.com/archives/freeipa-users/2011-June/msg00256.html. Online; Accessed: 2015-12-27.

[28] Red Hat: The world's open source leader. http://www.redhat.com. Online; Accessed: 2015-11-16.

[29] [RFE] Add a way to store and manage URLs/resources for applications. https://fedorahosted.org/freeipa/ticket/5030. Online; Accessed: 2015-11-16.

[30] RPM resource freeipa-server. http://rpmfind.net/linux/rpm2html/search.php?query=freeipa-server. Online; Accessed: 2015-11-16.

[31] Schema Specification. http://www.openldap.org/doc/admin24/schema.html. Online; Accessed: 2015-12-28.

[32] SDAP. https://fedorahosted.org/sssd/wiki/InternalsDocs#a5.2.SDAP. Online; Accessed: 2016-04-09.

[33] SSSD. https://fedorahosted.org/sssd/. Online; Accessed: 2015-12-29.

[34] SSSD InfoPipe responder. https://jhrozek.fedorapeople.org/sssd/1.12.0/man/sssd-ifp.5.html. Online; Accessed: 2015-12-29.

[35] SSSD Internals. https://fedorahosted.org/sssd/wiki/InternalsDocs. Online; Accessed: 2015-12-29.

[36] The Request for Feature Enhancement (RFE) Process for Red Hat product suite. https://access.redhat.com/solutions/73513#. Online; Accessed: 2015-11-16.

[37] trac – integrated SCM & Project Management. http://trac.edgewall.org/. Online; Accessed: 2015-11-16.

[38] Web UI. http://www.freeipa.org/page/Web_UI. Online; Accessed: 2016-04-10.

[39] What is a Directory Service? https://msdn.microsoft.com/en-us/library/windows/desktop/aa367035%28v=vs.85%29.aspx. Online; Accessed: 2015-12-27.

[40] Writing Apache Modules with Perl and C. http://docstore.mik.ua/orelly/apache_mod/24.htm. Online; Accessed: 2015-12-30.

[41] T. Berners-Lee, CERN, L. Masinter, Xerox Corporation, M. McCahill, University of Minnesota, and Editors. Uniform Resource Locators (URL). RFC 1738, IETF, December 1994.

[42] T. Berners-Lee, W3C/MIT, R. Fielding, Day Software, L. Masinter, and Adobe Systems. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, IETF, January 2005.

[43] T. Berners-Lee, W3C/MIT, R. Fielding, Day Software, L. Masinter, Adobe Systems, and Editors. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, IETF, January 2005.

[44] G. A. Champine, Jr. D. E. Geer, and William N. Ruh. Project Athena as a distributed computer system. *IEEE computer*, 23(9):40–51, Sept 1990.

[45] Ed. D. Crocker, Internet Mail Consortium, P. Overell, and Demon Internet Ltd. Augmented BNF for Syntax Specifications: ABNF. RFC 2234, IETF, November 1997.

[46] R. Fielding, UC Irvine, J. Gettys, Compaq/W3C, J. Mogul, Compaq, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, T. Berners-Lee, and W3C/MIT. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999.

[47] R.T. Fielding and G. Kaiser. The apache http server project. *Internet Computing, IEEE*, 1(4):88–90, Jul 1997.

[48] ITU-T. Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) . X. 690, ITU-T, July 2002.

[49] J. T. Kohl, B. C. Neuman, and T. Y. T'so. The evolution of the Kerberos authentication system. *Distributed Open Systems*, pages 78–94, 1994.

[50] Josh Lerner and Jean Tirole. The Economics of Technology Sharing: Open Source and Beyond. Working Paper 10956, National Bureau of Economic Research, December 2004.

[51] Robert Love. Get on the d-bus. *Linux Journal*, 2005(130):3, 2005.

[52] R. Moats and AT&T. URN Syntax. RFC 3986, IETF, May 1997.

[53] B.C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *Communications Magazine, IEEE*, 32(9):33–38, Sept 1994.

[54] Ellen Newlands. Who Goes There? Identity Management in Red Hat Enterprise Linux 7 Beta. http://rhelblog.redhat.com/2014/01/20/who-goes-there/, 2014. Online; Accessed: 2015-11-16.

[55] Dmitri Pal. Overview of Direct Integration Options. http://rhelblog.redhat.com/2015/02/04/overview-of-direct-integration-options/, 2015. Online; Accessed: 2015-11-16.

[56] Ed. R. Fielding, Adobe, Ed. J. Reschke, and greenbytes. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, IETF, June 2014.

[57] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[58] Vipin Samar. Unified login with pluggable authentication modules (pam). In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, CCS '96, pages 1–10, New York, NY, USA, 1996. ACM.

[59] V. Santuka, P. Banga, and B.J. Carroll. *AAA Identity Management Security*. Networking Technology: Security. Pearson Education, 2010.

[60] Eric Von Hippel. Learning from open-source software. *MIT Sloan management review*, 42(4):82–86, 2001.

[61] Wikipedia. Name service switch. https://en.wikipedia.org/w/index.php?title=Name_Service_Switch&oldid=670365245, 2015. Online; Accessed: 2015-12-29.