VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SSH PUBLIC KEY MANAGEMENT IN FREEIPA AND SSSD

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN CHOLASTA

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SPRÁVA VEŘEJNÝCH KLÍČŮ SSH V PROGRAMECH FREEIPA A SSSD

SSH PUBLIC KEY MANAGEMENT IN FREEIPA AND SSSD

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JAN CHOLASTA
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN ZELENÝ
SUPERVISOR

BRNO 2012

Abstrakt

SSH je jeden z nejpoužívanějších protokolů pro vzdálený přístup v Internetu. SSH je flexibilní a rozšiřitelný protokol, který se skládá ze tří hlavních součástí: SSH transportního protokolu, který obstarává důvěrnost, integritu a autentizaci serveru, SSH autentizačního protokolu, který obstarává autentizaci uživatelů a SSH spojovacího protokolu, který obstarává multiplexování více kanálů různých typů (interaktivní sezení, přesměrování TCP/IP spojení, atd.) do jednoho spojení. OpenSSH je jedna z nejrozšířenějších implemetací SSH. OpenSSH obsahuje SSH server, SSH klienty, generátor SSH klíčů a autentizační agent, který usnadňuje autentizaci pomocí veřejných klíčů. FreeIPA a SSSD jsou projekty poskytující centrální správu identit pro Linuxové a Unixové systémy. Tyto projekty sice v době psaní této práce přímou podporu SSH neobsahovaly, ale do jisté míry je ve spojení s OpenSSH používat možné bylo.

Abstract

SSH is one of the most frequently used remote access protocols on the Internet. SSH is flexible and extensible protocol, which consists of three main components: SSH transport layer protocol, which provides confidentiality, integrity and server authentication, SSH user authentication protocol, which provides user authentication and SSH connection protocol, which multiplexes multiple channels of different types (interactive sessions, TCP/IP forwarding, etc.) into one connection. OpenSSH is one of the most widespread implementation of SSH. OpenSSH contains a SSH server, SSH clients, a SSH key generator and an authentication agent, which eases public key authentication. FreeIPA and SSSD are projects which provide centralized identity management for Linux and Unix systems. These projects had no direct support for SSH at the time of writing of this paper, but nonetheless could be used in combination with OpenSSH to a certain degree.

Klíčová slova

SSH, autentizace veřejným klíčem, OpenSSH, správa identit, FreeIPA, SSSD.

Keywords

SSH, public key authentication, OpenSSH, identity management, FreeIPA, SSSD.

Citace

Jan Cholasta: SSH Public Key Management in FreeIPA and SSSD, diplomová práce, Brno, FIT VUT v Brně, 2012

SSH Public Key Management in FreeIPA and SSSD

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Zeleného. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Jan Cholasta
May 23, 2012

Contents

Chapter 1

Introduction

In organizations, it is desirable to maintain security information in a centralized manner, for ease of management and enforcement of policy. A broad range of software products is available which makes this possible. An example of such product is Microsoft Active Directory. In this paper we will focus on two open-source project, which provide centralized identity management in the Linux and Unix world, FreeIPA and SSSD. FreeIPA is an integrated solution combining several technologies; SSSD is one of these technologies. FreeIPA manages a domain with clients, servers and services, and SSSD provides access to the information in the domain.

SSH is a secure, universal remote access protocol, which provides confidentiality, integrity protection and authentication. Both the server and the user are authenticated, the server is authenticated using public key cryptography, the user may be authenticated using multiple authentication methods and public key authentication is one of them. In this paper we will discuss SSH and one of its implementations, OpenSSH, and how they can be used in an environment managed by FreeIPA.

In the second chapter of this paper we will discuss the base SSH protocol in detail. The different layers of the SSH protocol will be described, with special consideration given for the subject of public key authentication in SSH.

In the third chapter, we will discuss one of the most popular SSH implementations, OpenSSH. Each of the tools available in the OpenSSH suite will be outlined. The routines used to authenticate servers and users using public key cryptography will be described.

In the fourth chapter, FreeIPA, a centralized identity management solution for Linux and Unix systems, will be discussed. The architecture of this project will be outlined, as well as the current state of its support for SSH.

The fifth chapter will focus on SSSD, a service which provides access to different remote identity and authentication resources. We will discuss the architecture of SSSD and what it currently offers in terms of SSH-related features.

In the sixth chapter, the design of a FreeIPA and SSSD extensions allowing SSH public key management in FreeIPA domains and integration of OpenSSH into FreeIPA infrastructure will be described.

The seventh chapter will contain detailed description of implementation of the designed extension.

Finally, the eight chapter will be a short guide on building FreeIPA and SSSD with the extension and setting up a FreeIPA domain with OpenSSH integration enabled.

Chapter 2

SSH

SSH (*Secure Shell*) is a network protocol for secure access to remote services between two networked computers over an insecure network. SSH is typically used for access to shell accounts on Unix-like operating systems, but it also supports forwarding TCP ports and X11 connections and file transfer. There are two major versions of the protocol, referred to as *SSH-1* and *SSH-2*.

SSH provides confidentiality and data integrity by estabilishing a secure channel between the two communicating computers. Public-key cryptography is used to authenticate the server and can be used to authenticate the client-side user, which might also be authenticated by other means.

The first version of the protocol (SSH-1) was designed in 1995 by Tatu Ylönen, a researcher at Helsinki University of Technology, as a secure replacement for contemporary remote shell protocols, such as telnet, rlogin or rsh, which did not guarantee confidentiality nor provided strong authentication. In July 1995, Ylönen released his implementation of the protocol as freeware. In December 1995 he founded a company called *SSH Communications Security* to further develop and market SSH.

Over time, serious flaws in the design of SSH-1 were discovered [16][15][17] and work on the next generation of the protocol (SSH-2) had begun. A new Internet Engineering Task Force working group called *secsh* was created to design the second version of the protocol. In 2006, a series of RFCs (RFC 4250 to RFC 4256) were published by the working group, documenting SSH-2 as a proposed Internet standard.

SSH-2 improves security over SSH-1, as well as adds new features to the protocol. SSH-1 is now generally considered obsolete [3].

Further in this paper we will focus on SSH-2 only, SSH-1 will not be discussed anymore. When we refer to SSH, it will mean SSH-2 exclusively, unless stated otherwise.

2.1 The protocol

The protocol has been designed to be extensible. Algorithms, methods, formats and extension protocols are all identified by textual names in the protocol. DNS names are used to create local namespaces, where experimental or classified extensions may be defined without fear of conflict with other implementations. The base protocol has been designed to be as simple as possible and to require as few algorithms as possible.

There are three main components of the protocol:

- the *transport layer protocol* provides confidentiality, data protection, server authentication and optionally compression,
- the *user authentication protocol* runs on top of the transport layer protocol and provides user authentication,
- the *connection protocol* runs on top of the user authentication protocol and multiplexes multiple logical channels into a single connection.

New protocols may be defined and coexist with the protocols above [41].

2.1.1 Transport layer protocol

The SSH transport protocol is a secure low-level transport protocol. It provides strong encryption, server authentication and data integrity.

This protocol provides means to authenticate the server and only the server, user authentication is not part of it. A higher-level protocol can be used on top of this protocol for user authentication.

The protocol has been designed to be simple and flexibile and to keep the number of round-trips to estabilish a secure connection between two parties minimal. The algorithms used for key exchange, encryption, compression, hashing and message authentication are all negotiated when setting up the connection. It is expected that in most environments, the number of round-trips to do key-exchange, server authentication and a service request is 2. In the worst case, the number of round-trips is 3.

SSH can be used over any 8-bit clean, binary transparent transport. The transport should protect against transmission errors. When used over TCP/IP, the server usually listens on port 22 [42].

Connection setup

The client initiates the connection. When the connection has been established, both sides send an identification string, informing each other about the supported protocol version. If the protocol versions are compatible, the connection continues, otherwise it is terminated.

Key exchange

Encryption key exchange follows immediately after the identification string exchange. Key exchange may be started by either party by sending a list of supported algorithms for key exchange, encryption, compression, hashing and message authentication to the other party. Both parties have preferred algorithm for key exchange, encryption, compression, hashing and message authentication. A party may guess which key exchange algorithm the other party uses before the algorithm negotiation is done and send an initial key exchange packet according to the guessed algorithm. If the guess was right, the optimistically sent packet must be handled as the first key exchange packet. If the guess was wrong, the optimistically sent packet must be ignored and the appropriate side must send the correct initial packet.

Key exchange ends after the algorithms are negotiated by both sides informing the other side about taking new encryption keys into use. The message is sent using the old set of algorithms and encryption keys. All messages sent after this message must use the new

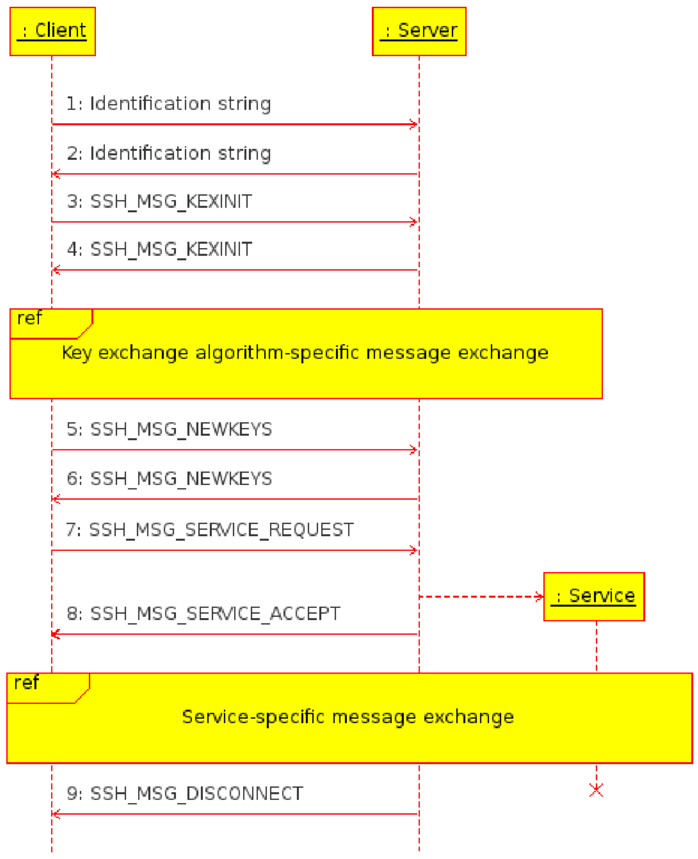


Figure 2.1: Message exchange in SSH transport layer protocol.

encryption keys and algorithms. When the message is received, all messages received after this message must use the new encryption keys and algorithms.

If compression has been negotiated, the contents of the packet will be compressed using the negotiated algorithm. When encryption is in effect, the contents of the packet must be encrypted with the negotiated encryption algorithm. Data integrity is protected by including a MAC (*Message Authentication Code*) with each packet that is computed from a shared secret, packet sequence number, and the contents of the packet with the negotiated message authentication algorithm.

Key exchange produces two values: a shared secret and an exchange hash. Encryption and authentication keys are derived from these values. The exchange hash from the initial key exchange is additionaly used as a session identifier, which is an unique identifier for the current connection. The session identifier never changes, even after another key exchange.

Key re-exchange

Either party may request key re-exchange by sending a key exchange initialization packet at any time except when already doing a key exchange. It is recommended to do key re-exchange after every gigabyte of transmitted data or after every hour of connection time, whichever comes sooner.

Server authentication

The server is authenticated during the initial key exchange. The client must have a priori knowledge of the server's public host key in order to authenticate it. All server hosts should have at least one host key. A server host may have multiple host keys, each using a different public key algorithm.

The authentication is either explicit or implicit. The authentication is explicit, if the key exchange messages contain a signature or other proof of authenticity of the server. The authentication is implicit, if in order to authenticate the server, the server has to prove that it knows the shared secret by sending a message and a corresponding MAC, which the client can verify.

Service request

After the initial key exchange, the client requests a service. The service is identified by a name. If the server rejects the service request, it must disconnect. If the server supports the service and permits the client to use it, it must respond with a message informing the client about accepting the service request.

There are 2 standard services: user authentication service (which uses the user authentication protocol – see subsection 2.1.2) and a connection service (which uses the connection protocol – see subsection 2.1.3).

Other messages

Either party may send a disconnection message, an ignored data message or a debug message at any time. The disconnection message causes immediate termination of the connection. The ignored data message contains arbitrary data and must be ignored. It can be used as an additional protection measure against advanced traffic analysis techniques. The debug message is used to transmit information that may help debugging.

If an unrecognized message is received, the implementation must respond with an unimplemented feature message and ignore the original message.

2.1.2 User authentication protocol

The SSH user authentication protocol is an universal user authentication protocol. It runs on top of the SSH transport layer protocol. This protocol assumes that the underlying protocol provides confidentiality and data integrity. It provides a secure tunnel for the SSH connection protocol.

When this protocol is started, it receives a session identifier from the lower-level protocol (the exchange hash from the initial key exchange). The session identifier uniquely identifies the session and is suitable for signing in order to prove ownership of a private key.

User authentication is driven by the server. It informs the client about what authentication methods the client can use at a given time. Authentication methods are identified by a name. The client may use any of the offered authentication methods in any order. This gives the server a complete control over the user authentication process, but also provides enough flexibility for the client to choose an authentication method which is supported or which is the most suitable for the user, if there are more methods offered by the server [39].

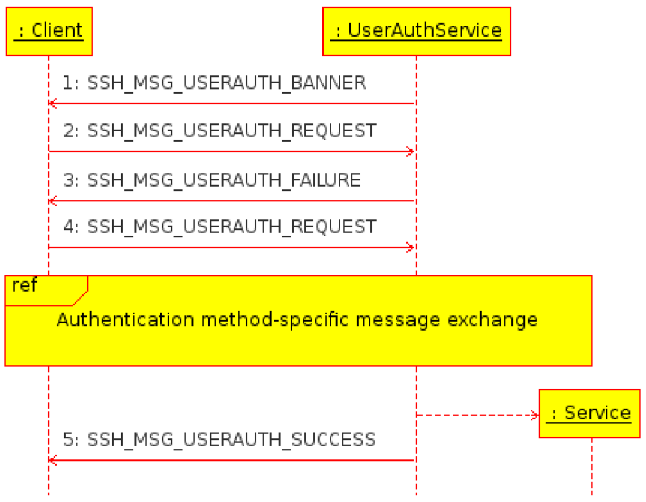

: Client
: UserAuthService
1: SSH_MSG_USERAUTH_BANNER
2: SSH_MSG_USERAUTH_REQUEST
3: SSH_MSG_USERAUTH_FAILURE
4: SSH_MSG_USERAUTH_REQUEST
ref
Authentication method-specific message exchange
: Service
5: SSH_MSG_USERAUTH_SUCCESS

Figure 2.2: Message exchange in SSH user authentication protocol.

Authentication request

User authentication is started by the client by sending an authentication request. The request contains the user name of the user being authenticated, the service name of the service to start after authentication, the name of the authentication method and authentication method-specific data. If the requested service does not exist, the authentication request must be rejected. If the requested user does not exist, the server may either disconnect, or send a bogus list of acceptable authentication method names, but never accept any of them.

The authentication request may require further messages to be exchanged. Whether that happens or not and what messages are exchanges depends on the authentication method used. If a new authentication request is received during this message exchange, current authentication is aborted and new one is started.

If the server rejects the authentication request, it must respond with an authentication failure message. The message contains a list of authentication methods, which might be used for further authentication requests and a boolean flag denoting partial success. Authentication methods used in previous successful authentications should not be included in the list. The partial success flag must be set if the authentication request was successful, but further authentication is necessary. The flag must be unset if the authentication request was rejected.

If the server accepts the authentication request, it must respond with an authentication success message. This message is not sent after each step in multi-method authentication, but only when the authentication is complete. This message must be sent only once. After it has been sent, any subsequent authentication request should be silently ignored.

Banner message

The server may send a banner message at any time after the user authentication protocol starts and before authentication is successful. This message contains text to be displayed to the client user before authentication is attempted.

Public key authentication

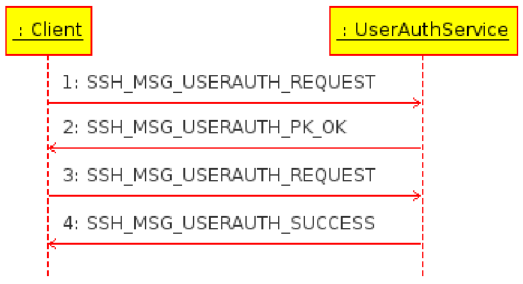

: Client
: UserAuthService
1: SSH_MSG_USERAUTH_REQUEST
2: SSH_MSG_USERAUTH_PK_OK
3: SSH_MSG_USERAUTH_REQUEST
4: SSH_MSG_USERAUTH_SUCCESS

Figure 2.3: Message exchange during user authentication with a public key.

The user authentication protocol allows a wide range of authentication methods to be used for user authentication. Since this paper's focus is on public key authentication, we will discuss only this single authentication method.

With this authentication method, the ownership of a private key serves as authentication. This method works by verifying a signature created with the user's private key. The server must check that the corresponding public key may be used to authenticate the user and that the signature is valid. If both these conditions are met, an authentication request must be accepted, otherwise it must be declined.

Private keys are usually stored in an encrypted form and the user must supply a passphrase to unlock the private key in order to create the signature. Even if the private key is not encrypted, the creation of the signature alone is an expensive computation. To avoid unnecessary processing and user interaction, the client may first query the server whether public key authentication would be acceptable using a special authentication request, which contains the public key, but not the signature. The server must respond to this message with either an authentication failure or with a message informing the client that it may authenticate using the given public key.

To perform the actual authentication, the client may send an authentication request which includes the signature created using the private key. The client may send this request without first checking if authentication with the given public key is acceptable. The signature is generated using the private key over the session identifier and the contents of the authentication request packet.

Any public key algorithm may be used for the authentication. The choice is not limited by what was negotiated during the initial key exchage. If the server does not support an algorithm, it should reject the authentication request.

2.1.3 Connection protocol

The SSH connection protocol has been designed to run on top of the SSH transport layer and user authentication protocol. It provides interactive sessions, remote execution of commands and forwarding of TCP/IP ports and X11 connections.

All terminal sessions, port forwarding, etc. are channels. Either side may open a channel. Multiple channels are multiplexed into a single connection.

Channels are identified by a number at each end. The number may be different on each side. A channel open request contains the sender's channel number. All other channel-related messages contain the recipient's channel number.

Channels are flow-controlled. No data may be sent into a channel until a message is received to indicate that window space is available [40].

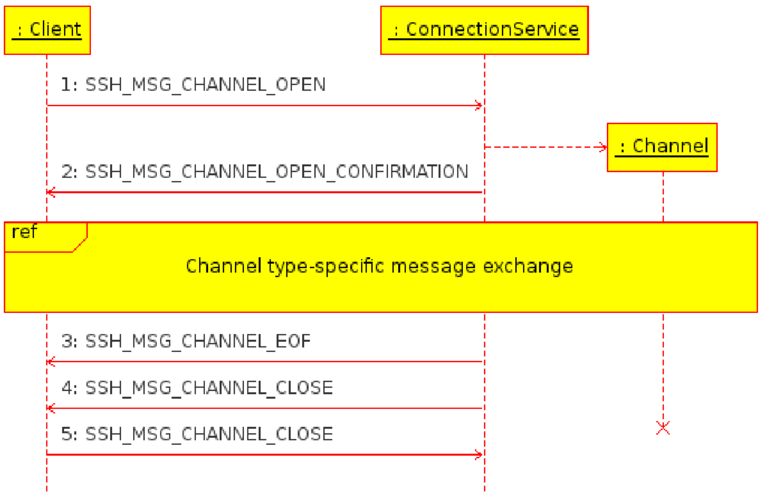


Figure 2.4: Message exchange in SSH connection protocol.

Global requests

There are several kinds of requests, which affect the state of the remote end globally, independent of any channel. Both parties may send a global request at any time. The request contains request type name, request-specific data and a boolean flag that indicates whether a response to the request is wanted. If the flag is unset, no response to the request will be sent. If the flag is set, the recipient responds with either a request success message or a request failure message. If the recipient does not recognize or support the request, it simply responds with a request failure message.

Opening a channel

When a party wishes to open a new channel, it allocates a local number for the channel and sends a channel open request to the other party. The request contains the requested channel type name, the local channel number, initial window size and maximum packet size. The other party responds with either a channel open confirmation, if the channel could be opened, or a channel open failure, if the channel could not be opened. The channel open confirmation message contains the local channel number, remote channel number and channel type-specific data. The channel open failure contains the reason code and description why the channel could not be opened.

The window size specifies how many bytes the other side may send before it must wait for the window to be adjusted. Both sides may adjust the window size by sending a message. After receiving this message, the recipient may send the given number of bytes more than what was previously possible.

Data transfer

Data transfer is done with a channel data message, which contains the transferred data. The maximum amount of data allowed is determined by the maximum packet size for the channel, and current window size, whichever is smaller. The window size is decreased by the amount of data sent. Both sides may ignore any data sent after the allowed window is empty.

Some channels may wish to transfer different kinds of data. Such data may be sent using an extended channel data message, which includes a channel type-specific integer that specifies the type of the data. Data sent with this message consume the same window as ordinary data.

Channel requests

Many channel types have extensions that are specific to that particular channel type. All channel type-specific requests are done using a channel request message. The request contains the request type name, request type-specific data and a boolean flag that indicates whether a response to the request is wanted. If the flag is unset, no response to the request will be sent. Otherwise, the recipient responds with either a channel success message, channel failure message, or a request-specific message. If the request is not recognized or supported by the channel, channel failure is returned.

Closing a channel

When a party will no longer send more data to a channel, it should send a channel EOF message. No response is sent to this message. The channel remains open after sending this message and data may still be sent in the other direction.

When either party wishes to close the channel, it sends a channel close message. When this message is received, a party must respond with a channel close message, unless it has already been sent for the channel. The channel is considered closed for a party when it has both sent and received channel close message. The local channel number is freed after the channel is closed and may be used again. A party may send a channel close message without first sending a channel EOF message.

2.2 Protocol extensions

Since the protocol was first introduced, a number of extensions was created and is commonly used today. Some of the well-known extensions are:

- storage of host public key fingerprints in DNS SSHFP records, defined in RFC 4255 [22],
- generic interactive challenge-response user authentication method, defined in RFC 4256 [5],
- GSSAPI key exchange and user authentication method, for use with Kerberos and other authentication protocols, defined in RFC 4462 [11],
- usage of elliptic curve cryptography for public keys and key exchange, defined in RFC 5656 [23],
- *SFTP* (the *SSH file transfer protocol*), which provides access to a remote filesystem using SSH, defined in an IETF draft [6].

Chapter 3

OpenSSH

OpenSSH is a set of computer programs that implement the SSH protocol (both SSH-1 and SSH-2). It is developed as part of the open-source OpenBSD project [30] and has been ported to various platforms besides OpenBSD, including other Unix and Unix-like operating systems and Microsoft Windows. It is released under the BSD license, like the rest of the OpenBSD project.

OpenSSH is a fork of OSSH, which was created in 1999 by Björn Grönvall. OSSH itself is a fork of the original SSH software by Tatu Ylönen. Because more and more licensing restrictions were introduced with each new release of the original SSH software, the version that OSSH was derived from is ssh 1.2.12, which was released in 1995 [38]. It was the last release that used a license free enough to be used as the base of an open-source derivative.

OpenSSH was created in 1999, two months before OpenBSD 2.6 release. SSH support was planned for OpenBSD 2.6 and after a rapid development phase, OpenSSH 1.2.2 was released and included in OpenBSD 2.6. This version featured many feature improvements and bug fixes over ssh 1.2.12, most notably the replacement of non-free parts with free alternatives, support for Kerberos IV authentication and for one-time password authentication with S/KEY.

After the release of OpenBSD 2.6, work on implementing support for SSH-2 into OpenSSH had started. The work was completed in OpenSSH 2.0, which was shipped with OpenBSD 2.7 in 2000. This version supported both SSH-1 and SSH-2. Support for server-side of SFTP, the SSH file transfer sub-protocol, was added in OpenSSH 2.3.0. SFTP client was added in OpenSSH 2.5.0 [32].

A portable version of OpenSSH is maintained along the mainline version of OpenSSH, which is developed exclusively for OpenBSD.

The current version of OpenSSH is 5.9, released on September 6, 2011 [31].

3.1 Components

OpenSSH implements the SSH protocol, the SFTP sub-protocol and extensions to both of them. The implemented extensions include a tunelled VPN (*Virtual Private Network*) operating on OSI layer 2 or 3, new algorithms for key exchange, data integrity and public key authentication and support for certificate authentication of users and servers (these are OpenSSH-specific certificates, not to be confused with X.509 certificates).

The main components of OpenSSH are:

- *sshd*, the SSH server implementation. It is usually run in one system-wide process, which listens for connections from clients. Each client connection is handled in a separate process, which is forked from the main process when the client connects.
- *ssh*, the SSH client implementation. It can be used to access a remote shell (like rlogin), to run commands on a remote machine (like rsh), to forward a TCP port to or from a remote host or to set up a VPN between the client and a remote host. Additional SSH client tools are available in OpenSSH: *scp*, a tool for copying files between remote hosts (like rcp) and *sftp*, a command-line based SFTP client implementation.
- *ssh-agent*, the user authentication agent. It is an utility which eases user authentication by holding user's private keys ready in memory. This avoids the need to enter the passphrase to unlock the key every time it is used. An accompanying utility, *ssh-add*, is provided for manually adding keys to the authentication agent.
- *ssh-keygen*, a tool to generate private and public keypairs for users and hosts. It supports both plain keys and OpenSSH certificates. It also supports inspecting the keys and generating DNS SSHFP records for DNS fingerprint storage from the keys.

3.2 Public key authentication

OpenSSH supports public key authentication with either plain public keys or with OpenSSH certificates. We will discuss authentication with plain public keys only.

OpenSSH supports the DSA and RSA public key algorithms, as defined in RFC 4253 [42], as well as the newer ECDSA algorithm, as defined in RFC 5656 [23]. These algorithms are allowed for both user and host keys.

3.2.1 Server authentication

In SSH, a client must have a priori knowledge of server's public host key in order to verify its identity [41]. OpenSSH client maintains a database of servers and the associated public host keys. Whether an unknown server is to be trusted or not is in the hands of the user. When the client first connects to a server, a message containing a fingerprint of the server's public host key is displayed to the user and the user is asked to verify the identity of the server. If the user verifies the identity, the server's host name and IP address and the public host key are added to the database. On subsequent connections, the server's identity is verified against the database. This includes checking the public host keys as well as the IP address.

It is possible to configure OpenSSH to automatically trust all unknown servers, or to automatically trust those servers, whose public host key fingerprint matches the fingerprint stored in DNS SSHFP record for the server's host name. However, using this configuration might be a security risk.

3.2.2 User authentication

A server must have a priori knowledge of user's public keys in order to verify the identity of the user. OpenSSH server reads user's public keys from a database in the user's home directory. The database is operated manually by the user and/or by an administrator.

If the user's public key is not present in the database, the authentication attempt fails. However, the user might still authenticate using other authentication methods, if the server allows it.

Chapter 4

FreeIPA

FreeIPA is an integrated solution for centralized management of security information in organizations. It acts as a domain controller for domains where Linux and Unix servers and clients share centrally-managed services [14].

FreeIPA uses existing technologies such as Linux (namely Fedora), 389 Directory server, MIT Kerberos 5, Dogtag certificate server, Apache HTTP server, BIND DNS server and a NTP server and combines them in one unified environment. It provides command-line and web-based tools for administrators to manage the domain. FreeIPA is an open-source project released under the GPL license. Its development is backed up by Red Hat [28].

FreeIPA was created in 2007 by Red Hat. The goal of the project was to provide an easy-to-use centralized management of identities for Linux, combining directory server for user information storage and Kerberos for authentication and single sign-on.

Version 1 was released in 2008. It featured tools to automate the configuration of the directory server and the Kerberos server and set up replication. The provided command-line and web-based tools allowed administrators to manage users and groups.

Version 2 was released in 2011. Many new features were added in this release, most notably the integration of the certificate server and the DNS server, support for managing hosts, netgroups and automount maps and authentication policy using host-based access control. An extensible framework was provided for developers to easily add new functionality through plugins [29].

Version 3 is currently in development and is scheduled to be released in 2012.

4.1 Architecture

The two major parts of FreeIPA are the FreeIPA server and the FreeIPA client. FreeIPA domain consists of one or more FreeIPA server hosts and a FreeIPA client on each managed host.

4.1.1 Server

At the very core, FreeIPA uses its own modular Python framework called *ipalib*, which allows it to be easily extensible through plugins. All of the backend and frontend functionality is implemented using such plugins.

FreeIPA provides a management API, which is available on the server through XML-RPC and JSON-RPC RPC interfaces. The management tools use these interfaces to communicate with FreeIPA servers.

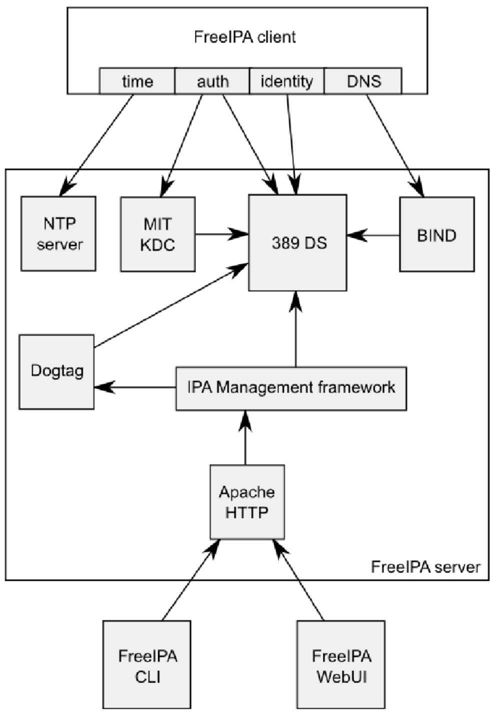


Figure 4.1: High-level diagram of FreeIPA architecture.

Backend

The backend of FreeIPA is based on the following open-source technologies:

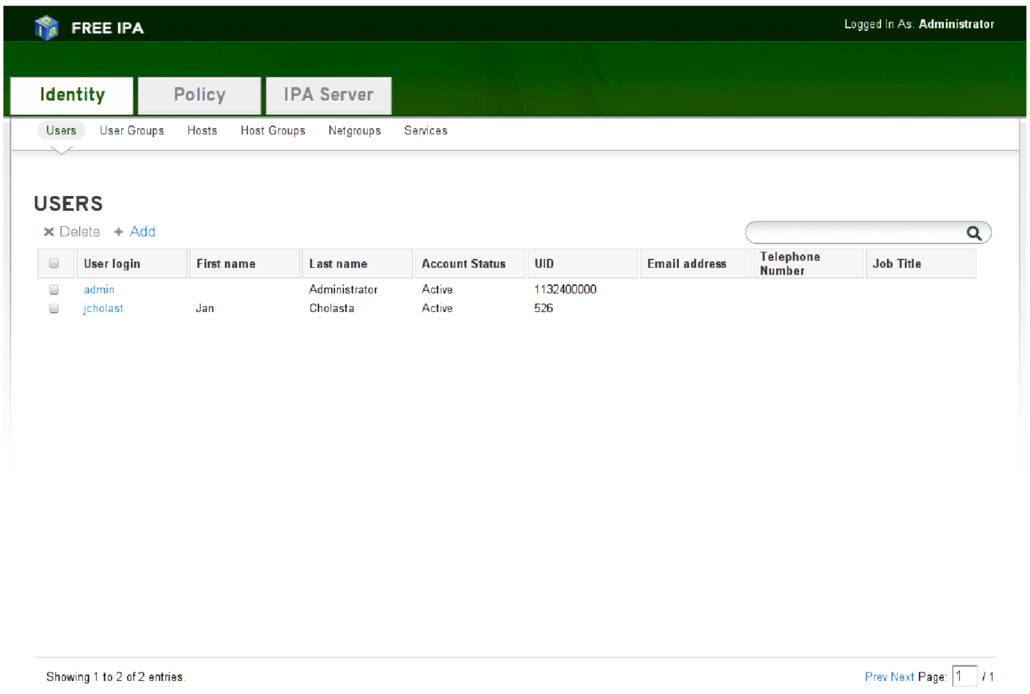
- *Fedora*, a Linux-based operating system [27]. It is used as the software platform for FreeIPA.
- *389 Directory Server* (formerly *Fedora Directory Server*), a LDAP directory server [24]. It supports multi-master replication, fine-grained access control, secure authentication and transport using SSL/TLS or SASL, plugin interface and many other features [18]. It is used as a storage backend for virtually all the information in a FreeIPA domain. Standard LDAP object classes are used in the schema as much as possible, to ensure interoperability. FreeIPA extends the directory server with plugins for host enrollment and other FreeIPA-specific features.
- *MIT Kerberos 5*, an implementation of the Kerberos network authentication protocol [19]. It provides cryptographically strong authentication of users, hosts and services and single sign-on.
- *Dogtag Certificate System*, a public key infrastructure management solution [26]. It provides the certificate authority functionality in FreeIPA. Certificates can be issued for hosts and services in a FreeIPA domain and used to authenticate them.
- *Apache HTTP Server*, a HTTP server [25]. It is used to provide the XML-RPC and JSON-RPC interfaces as well as serve the web-based management tool.
- *BIND*, a DNS server and resolver fully compliant with the published DNS standards [13]. It provides the DNS service in a FreeIPA domain.

FreeIPA supports multi-master replication for fault-tolerance and load balancing between more servers. A FreeIPA server can be cloned into a replica and the domain information is automatically synchronized between them. Clients can communicate with replicas the same way as they would with the original server. It is possible to create complex replication setups, with replicas of replicas and replication between more FreeIPA servers [14].

Frontend

The frontend of FreeIPA consists of a number of plugins, where manageable objects and the commands to manage them are defined. Each object plugins defines an object type (such as user) along with its attributes (such as username), which represent a class of LDAP entries stored on the directory server. Command plugins define the commands to query, create, update and delete objects of a particular object type. All the commands for all the object types combined form the management API.

Management tools



User login	First name	Last name	Account Status	UID	Email address	Telephone Number	Job Title
admin		Administrator	Active	1132400000			
jcholast	Jan	Cholasta	Active	526			

Showing 1 to 2 of 2 entries.

Figure 4.2: FreeIPA web-based management tool.

FreeIPA provides command-line and web-based tools for managing the information in the domain.

The command-line tool is a powerful tool that can be used not only for manual management of the domain, but also for automation of repeatedly performed management tasks without manual intervention. It supports adding custom attributes to entries, if they are supported in the schema. It uses the XML-RPC interface to access the API, or, when invoked on the FreeIPA server, accesses the API directly in Python code.

The web-based tool is a graphical user interface tool, which is intuitive to use and easy to orientate in. It visualises the objects and the relationships between them. It uses the JSON-RPC interface to access the API. By default, it uses Kerberos for authentication, but it can be configured to use password authentication, so that it can be accessed from outside the FreeIPA domain [14].

4.1.2 Client

FreeIPA provides installation tools to configure a host as a FreeIPA client. As part of the installation process, the host is enrolled to the FreeIPA domain and becomes a managed host. It is configured to be able to access the identity information and policy settings in the domain. SSSD [37] is used on the client to provide access to the information in the domain [14].

4.2 SSH support

Currently there is no direct support for SSH in FreeIPA. SSH public keys cannot be stored for users and/or hosts in FreeIPA. OpenSSH (or any other SSH implementation) is not configured as part of the client installation process.

Users may use SSH client to log into FreeIPA-managed hosts using authentication methods that FreeIPA supports and that are not specific to SSH (password authentication, Kerberos authentication), but the SSH server must be manually configured on each of these hosts to enable this.

Chapter 5

SSSD

SSSD (*System Security Services Daemon*) is an extensible service which provides access to different identity and authentication remote resources and caching of information for offline access. It is an open-source project developed by Red Hat, released under the GPL license [37].

SSSD was created in 2008 by Red Hat. The goal was to create a collection of components for obtaining and caching of the domain information from FreeIPA servers and providing it to the underlying system [20].

The first SSSD release was in 2009. It featured a local database backend, LDAP authentication backend, caching and PAM and NSS interfaces. SSSD quickly reached version 1.0.0, which was released in December 2009. This version featured a complete LDAP backend, Kerberos V authentication backend and FreeIPA backend with host-based access control.

The current version of SSSD is 1.7.0, released on December 12, 2011.

SSSD is used as the default authentication method for remote authentication in Fedora since the Fedora 13 release [8]. Other Linux-based operating systems have started adopting it as well.

5.1 Architecture

SSSD is designed to handle multiple domains. Each of these domains may use different backends for identity information, authentication, access control and changing of passwords. Each of the backends may use different remote resource. SSSD maintains a database for each domain where user information is cached for offline access. Backend-specific extended information may be stored for each user.

SSSD interfaces with the underlying operating system via NSS (*Name Service Switch*) and PAM (*Pluggable Authentication Modules*) modules. This gives the operating system access to the identity information and authentication services provided by the remote resources in each domain.

SSSD maintains a special local domain for local user storage, as a complement or replacement of the standard Unix user database. Advanced features available for all domains, such as nested groups, are available for the local domain as well. SSSD provides a set of tools similar to standard Unix tools for creating, modifying and deleting users from the local database.

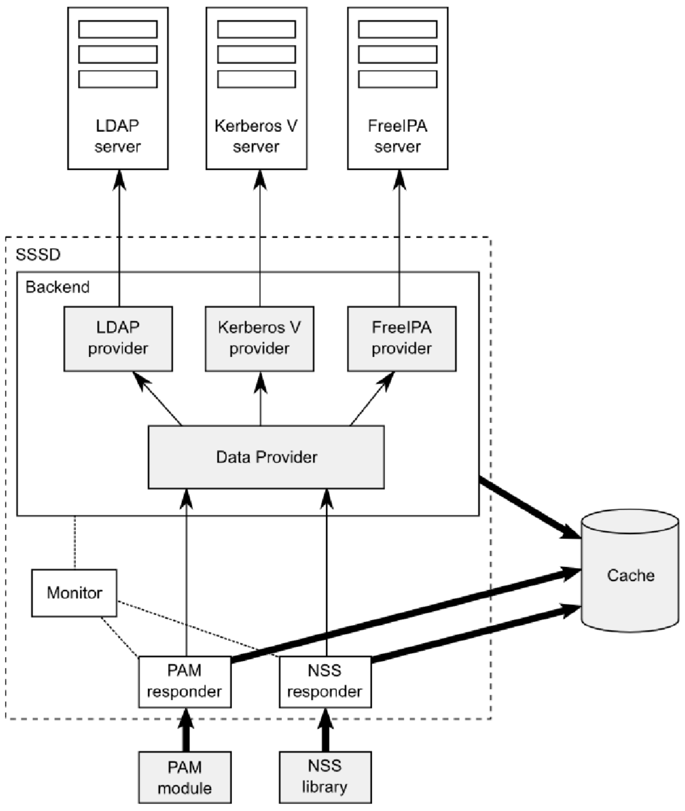


Figure 5.1: High-level diagram of SSSD architecture.

SSSD runs a set of processes, each handling a specific task:

- *monitor* is the supervising process, which watches over the other processes and starts and restarts them as needed,
- *data provider* is the process which caches the information from other providers and automatically queries the providers to obtain up-to-date information and update the cache with it,
- *provider* is a backend process which handles identity and authentication information exchange with a remote resource,
- *responder* is a process which handles communication with the underlying operating system services.

All of the processes are single-threaded and use an event loop for pseudo-concurrence [10].

Each provider implements routines to access identity information, authentication, access control and changing of passwords (or a subset thereof) from a remote database. Currently there are providers for LDAP, Kerberos V and FreeIPA and a special provider, which proxies access to foreign NSS and PAM modules.

The PAM and NSS module communicate with SSSD using PAM and NSS responder, respectivelly.

5.2 SSH support

Currently there is no support for SSH in SSSD. None of the providers manage SSH public keys for users and/or hosts. Beyond that, it is not possible for a SSH implemetation to access public keys through SSSD, as PAM is not suitable for SSH public key authentication (public key authentication is handled by the SSH server itself, not the underlying operating system) and NSS does not provide means to access user nor host SSH public keys.

The SSH implementation may only use the identity information and authentication methods provided by the SSSD NSS and PAM modules.

Chapter 6

Integrating OpenSSH into FreeIPA and SSSD

As discussed earlier in this paper, neither FreeIPA nor SSSD provide tighter integration with OpenSSH (or any other SSH implementation) than what is available through standard operating system interfaces. In order to improve the situation, I have designed an extension to these applications, which will allow SSH public key management for both hosts and users in FreeIPA domains and integrate OpenSSH into FreeIPA.

Before I begin describing the design, let me outline the basic goals I aim to accomplish. The extension will:

- allow storing user and host SSH public keys centrally in a FreeIPA domain,
- provide a management interface to allow administrators to add and delete the public keys,
- automatically distribute the public keys to FreeIPA client hosts in the domain,
- provide an interface between FreeIPA and OpenSSH on FreeIPA client hosts to allow OpenSSH to use the public keys.

The extension shall be a first-class citizen in FreeIPA. To fit in FreeIPA ecosystem nicely, it should satisfy the following requirements:

- store the public keys on LDAP directory server,
- use ipalib to build the management interface,
- use SSSD on the client side for communication with FreeIPA servers,
- build the FreeIPA–OpenSSH interface on top of SSSD,
- configure OpenSSH as part of FreeIPA client installation process.

6.1 OpenSSH-LPK

There is an existing project, OpenSSH-LPK [1], which provides an OpenSSH patch that makes it possible to use public keys stored on a directory server using a custom LDAP schema. The schema allows multiple public keys in OpenSSH authorized keys format to be stored in user entries. `sshd` can then be configured to look for user public keys on one or more directory servers in a specified directory subtree. The patch also implements simple access control based on group membership of users [2].

OpenSSH-LPK mets all of the basic goals for the extension except for providing a management interface. I have considered using OpenSSH-LPK as a base for the extension, but there is a number of issues that would need to be resolved first:

- OpenSSH-LPK requires patching `sshd`, as the functionality is not available in OpenSSH upstream. Most operating systems do not ship OpenSSH with the patch included. This means that in many cases, users would have to apply the patch on OpenSSH sources and build OpenSSH themselves.

- There is no support for host public keys, only for user public keys. While extending the LDAP schema to support host public keys would be rather straightforward, extending ssh with LDAP support would require a non-trivial amount of changes, comparable to the amount of changes to sshd in the supplied patch.

- OpenSSH-LPK patched sshd talks directly to the directory server, bypassing SSSD. No caching of LDAP search results is done, which might have bad impact on overall performance in larger environments. `sshd` would need to be further extended to communicate with the directory server through SSSD.

- Access control is based solely on group membership of users. FreeIPA uses *HBAC* (*Host-Based Access Control*), which allows fine-grained access control based on user identity and group membership, target host identity and host group membership and type of the requested service. `sshd` would need to be changed to use SSSD for access control, as it implements HBAC.

Because none of these issues is easily resolvable and would require rewriting most of OpenSSH-LPK from the grounds up, I have decided not to use it as a base for my work, but rather to design and implement the extension from scratch.

6.2 SSH public keys on FreeIPA server

FreeIPA uses 389 directory server (see subsection 4.1.1) to physically store the information in the domain and SSH public keys should be no exception to this practice. For the needs of the extension, an LDAP schema for storing user and host SSH public keys will be defined, along with access control rules restricting who can add and delete public keys of what users and/or hosts.

6.2.1 LDAP schema

In short, LDAP schema consists of object class definitions and attribute type definitions. Object class defines what attributes a directory entry must or may contain. Each directory entry may have one or more object classes. Attribute type defines what kind of values are allowed in attributes of this type and whether such attribute may occur multiple times in a single directory entry [21].

We need to store SSH public keys for users and hosts, so two new object classes will be defined: one representing SSH hosts and the other representing SSH users. Both object classes will have a single attribute representing the public key. As neither users nor hosts are required to have a public key, the attribute shall be optional. It is possible for both hosts and users to have multiple public keys, so the attribute will be multi-valued. There are 3 formats that can be used to represent a public key in the attribute:

- raw public key blob, as per RFC 4253 [42],
- textual representation of public key in OpenSSH format [34],
- textual representation of public key in RFC 4716 format [7].

I have decided to use the raw public key format in the attribute, as it is the most compact format and does not allow any additional data besides the public key itself.

6.2.2 Access control

389 directory server allows fine-grained access control to specific entries and attributes based on various criteria. Access control rules are defined in *ACIs* (*Access Control Instructions*). ACIs for an entry are stored within the entry in a special attribute and apply to the entry, its child entries and attributes [21].

FreeIPA allows managing access control rules to a certain extent. There are several object types which allow manipulating ACIs and assigning rights to users [14]:

- *permission* grants access to read or write attributes or to add or delete entries in a specific target – the target can be all entries of a given object type, all entries in a given subtree, all entries that match a given LDAP filter, all members of a group or a single specific group,
- *privilege* combines permissions needed for a specific task into a single logical unit,
- *role* assigns a set of privileges to specific users,
- *selfservice* grants users access to read or write their own attributes,
- *delegation* grants members of one group access to read or write attributes of members of an other group.

As the term suggests, a *public* key should be publicly accessible and readable by anyone. Write access on the other hand should be allowed only under certain conditions to a limited set of entities.

Members of the administrators group have unlimited access to the whole directory by default. Administrators should have the ability to delegate the right to manage SSH public keys to other users. Two new permission objects, granting write access to the SSH public key attribute of users and hosts respectively, will be created for this purpose. Additionally, users should have the ability to manage their own public keys, so a new selfservice object will be needed. Hosts must be able to write their own SSH public keys too, so that they can be modified in FreeIPA client installation. A new ACI will be added to enable this.

6.3 Management interface

The management interface of FreeIPA can be extended with new object types and commands using plugins (see subsection 4.1.1). In order to allow management of SSH public keys, the existing plugins for management of users and hosts need to be extended with support for the new LDAP object classes and attribute type.

Since there is just a single new attribute for both users and hosts, new management plugin is not needed. The user and host plugins will be enhanced with support for the SSH public key attribute. This will allow administrators to specify public keys when adding or modifying users or hosts and users to modify their own public keys.

When FreeIPA is updated from a previously installed version, existing user and host entries on the directory server will not have the SSH object classes set. This means that SSH public keys can't be added to them by simply setting the public key attribute, as that would be an object class violation. The SSH user and SSH host object classes must be added to the user and host entry respectively, when a public key is being added to the entry for the first time.

The public key is a binary blob and usually quite long. When interacting with a user, it is common to display a fingerprint of the public key instead of the full public key blob. The management interface should support this by displaying the fingerprint by default and the full public key only when requested. A new attribute will be added to the user and host plugins for this purpose. The attribute will be virtual, its values will be generated from the values of the public keys attribute and will not be physically stored on the directory server.

Host public key fingerprints can be stored in DNS in SSHFP records for the host. Updating the SSHFP records manually would be a burden for the administrator, so automatic management of SSHFP records shall be supported in the host plugin. When a host is added to FreeIPA with public keys set, SSHFP record for the host will be automatically created if possible. Similarly, when a host is modified, the SSHFP records will be updated and when a host is deleted, the SSHFP records will be deleted as well.

6.4 SSH public keys in SSSD

On the client side of FreeIPA, SSSD is used to communicate with FreeIPA servers and to provide authentication, authorization and other services to the underlying operating system. For SSH public key support, SSSD needs to be extended to allow retrieval of the public keys from FreeIPA servers, caching them for offline use and providing them to other applications.

6.4.1 Backend

The SSSD backend consists of the data provider and remote service providers (see section 5.1). The data provider handles cache update requests and redirects them to the proper remote service provider. When a remote service provider receives the request, it attempts to retrieve the requested data from the remote service and if it is successful, it updates the cache. The data provider will be extended to support requests for user and host SSH public keys. The FreeIPA remote service provider will be extended to support storing the public keys in the cache.

Adding support for user public keys should be rather straightforward. The data provider handles request for user and other account data in its account info handler. the LDAP remote service provider, which the FreeIPA provider is based on, allows mapping of LDAP attributes to cache attributes. The logic for retrieving the attributes and storing them in the cache is internal to the LDAP provider. This means that the only modification needed to support user SSH public keys in the SSSD backend is adding the public key attribute to the user attribute map.

Supporting host public key needs more complex changes, though. Neither the data provider nor the LDAP provider have support for requesting and caching of host data. To rectify this, a new request handler type for SSH host information requests needs to be added to the data provider. The FreeIPA provider will be extended to support this new handler type and retrieve host public keys from FreeIPA servers and store them in the cache when requested.

6.4.2 Responder

SSSD clients, which provide the interface between SSSD and the operating system, communicate with SSSD through responder processes (see section 5.1). Each responder listens on a socket for commands from associated clients and is responsible for issuing data provider requests and returning the correct data from cache to the clients.

The clients must be able to acquire SSH public keys for user and hosts from SSSD. Two new client commands will be added for this purpose and a responder will be created for dispatching the commands. Both commands must accept a name and return a (possibly empty) list of public keys to the client. When dispatching a command, the responder will first request cache update on the data provider and then read the public keys from the cache and send them back to the client. This order of events ensures that up-to-date public keys are always returned to the client, unless the remote service (FreeIPA) is offline.

It is possible to have multiple domains configured in SSSD. Sometimes it might be necessary to restrict the search to a specific domain. To allow this, the name argument should be accepted in two forms: the unqualified form where only the name is specified, which will result in a search over all domains, and the qualified form where the name and domain are specified, which will result in a search only in the given domain.

6.5 Interface between SSSD and OpenSSH

One of the most important things to do, if not the most important, is to provide an interface for OpenSSH, which it will use to get public keys from SSSD. Because there are two kinds of public keys, host and user public keys, two interfaces are actually needed. The host public key interface will be used on the SSH client side, by `ssh`, the user public key interface will be used on the SSH server side, by `sshd`. Both shall be implemented as SSSD clients and communicate with the SSH responder of SSSD.

6.5.1 Host public key interface

OpenSSH loads known host public keys from a file called `known_hosts`. A per-user `known_hosts` file is automatically managed by `ssh` (see subsection 3.2.1), but it can also be configured to use a system-wide `known_hosts` file. When the system-wide `known_hosts` file is used, `ssh` looks for a host's public keys in it before looking in the per-user `known_hosts` file. In order to provide FreeIPA host public keys to `ssh`, SSSD can either manage a system-wide `known_hosts`, or use a functional equivalent of it.

The most primitive way of automatically managing the `known_hosts` file is to periodically update it with public keys of all hosts in the domain. This solution would be very easy to implement, but there is one major shortcoming to it: it is not scalable at all. Even if only the differences between the client local state and domain global state are transmitted to clients in each update, it would still be an enormous load on FreeIPA servers in domains with hundreds or more hosts. This kind of solution should not be considered at all. Instead of doing updates periodically and in bulk, they should be done on demand and for specific hosts.

The next obvious thing to do would be to modify `ssh` to support getting known host public keys not only from the `known_hosts` file, but also from SSSD using inter-process communication. This way the condition of doing updates on-demand and for specific hosts would be satisfied, as SSSD would be contacted before the connection to a specific SSH server is attempted. However, this solution has the same disadvantage as OpenSSH-LPK: it requires patching OpenSSH. This should be avoided and considered only when there is no other option available, because it might negatively impact adoption of the extension.

OpenSSH supports verification of host public key authenticity using DNS SSHFP records [33]. If this feature was used, everything that would have to be done would be to enable it in `ssh` and to automatically manage SSHFP records on FreeIPA DNS servers for all hosts in the domain. As nice as this solution might sound, there are some shortcomings to it:

- DNS service in FreeIPA is optional and doesn't have to be installed at all,
- DNSSEC is not yet supported in FreeIPA DNS service, which makes the solution vulnerable to DNS spoofing attacks,
- when public keys of a host are changed, they may not be synchronized with DNS SSHFP records on non-authoritative DNS servers for a short period of time,
- only DSS and RSA SSH public key algorithms are supported in DNS SSHFP records.

Despite all the shortcomings, this solution is viable, it just cannot be used in all environments. Once DNSSEC support is available in FreeIPA, it should be implemented in the extension, at least for the sake of other software packages (any besides OpenSSH) which implement the SSH protocol.

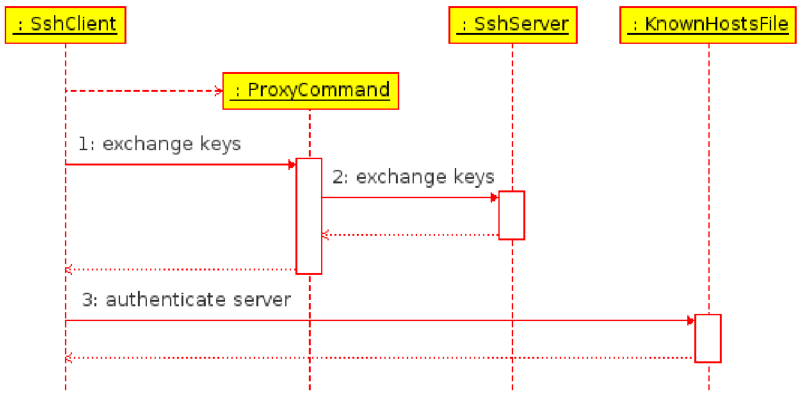


Figure 6.1: Connecting to a SSH server behind a proxy.

To be able to access SSH servers behind a proxy, `ssh` provides a configuration option which allows specifying a custom command that is used for connecting to such servers. Instead of communicating with a server directly, `ssh` executes the command and pipes the connection through the command's standard input and output. The point at which the command is executed has one nice characteristic: it occurs after the host name of the server is known, but before attempting to authenticate the server. We could take advantage of this and create a fake proxy command, that would first get public keys of the server from SSSD, update the `known_hosts` file with the public keys and then estabilish the connection, without actually doing any proxying. This solution does not need patching OpenSSH, would allow the updates to be done on demand and for specific hosts and should work in all environments. This makes it the perfect candidate for the host public key interface, so it will be implemented in the extension.

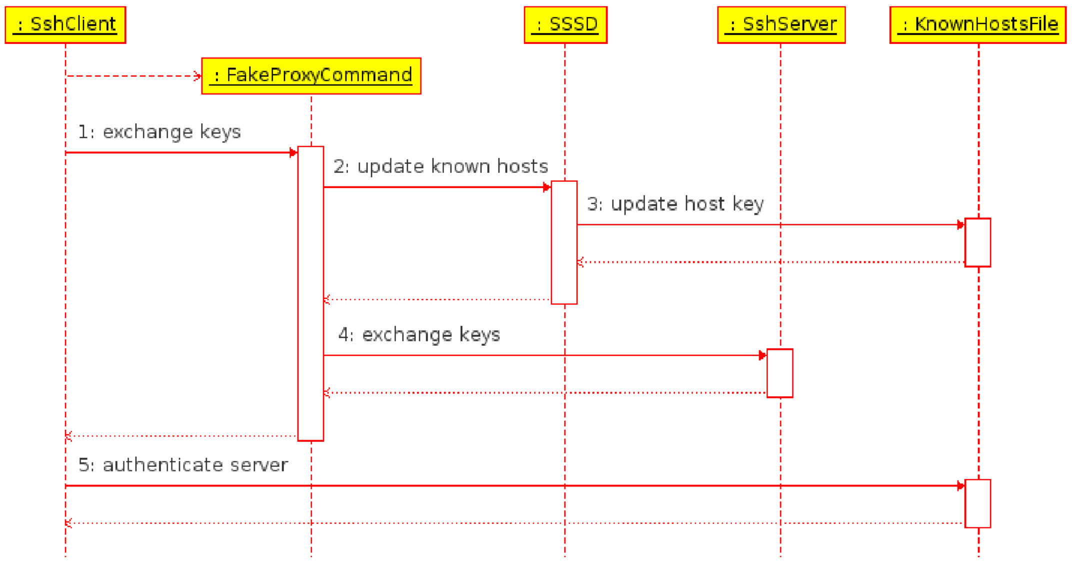


Figure 6.2: Connecting to a SSH server using SSSD fake proxy command.

In `ssh`, the server can be specified by a fully-qualified domain name, relative domain name or IP address. The server's public key is expected to be identified by this string in the `known_hosts` file. In FreeIPA and SSSD, hosts are identified by their fully-qualified domain name. The fake proxy command must perform reverse DNS lookup on IP addresses and canonicalize relative domain names to get the fully-qualified name of the server before it can get the public keys from SSSD, but it also must make sure that the original identification string is used in the generated `known_hosts` file.

When a SSH server is behind a proxy, `ssh` must use a real proxy command to connect to it. Because `ssh` cannot use both the fake proxy command and the real proxy command at the same time, the fake proxy command itself must be able to use the real proxy command to connect to the server. This way `ssh` will still can use the fake proxy command to update the `known_hosts` file and estabilish the connection, but the fake proxy command will use the real proxy command to connect to the server instead of connecting to it directly.

6.5.2 User public key interface

User public keys are loaded from a file called `authorized_keys` [34]. It is a per-user file which is managed by the user (see subsection 3.2.2). The user public key interface can be based either on managing `authorized_keys` files in SSSD, or possibly on transferring the public keys from SSSD to sshd by other means.

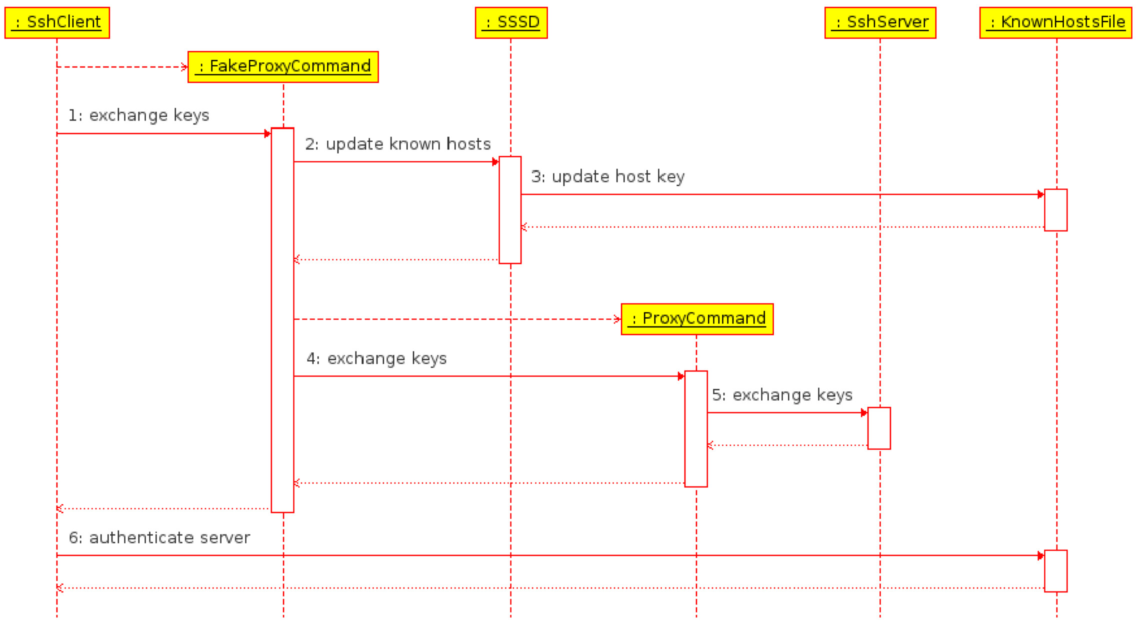


Figure 6.3: Connecting to a SSH server behind a proxy using SSSD fake proxy.

Similarly to how the host public key interface for `ssh` could be done, SSSD could update `authorized_keys` files for all users in the domain, or `sshd` could be patched to get the public keys directly from SSSD. These solutions carry all the disadvantages of their host public key counterparts. Updating `authorized_keys` periodically is actually even worse, as there is one such file to be updated for each user in the domain. Unlike `ssh`, there is no `sshd` configuration option which could be used to update `authorized_keys` file of a specific user at the right moment.

Because of the unfavorable situation, the only solution that can be used is patching `sshd`. Fortunately, there is already a `sshd` patch that can be used for getting public keys of a user from SSSD. The patch adds an option to use a custom command for getting the public keys. The patch is relatively small and simple and some operating system distributors ship OpenSSH with it included (e.g. Fedora and RHEL). With the patch applied, `sshd` executes the command on each user public key authentication attempt. The command's output should contain the user's public keys in the same format that `authorized_keys` files use. The user's `authorized_keys` file is then used only if no public key in the command's output matches the public key the user is attempting to authenticate with.

A command that gets user public keys from SSSD and outputs them in `authorized_keys` format will be implemented in the extension for the user public key interface. It will obviously work only with patched OpenSSH.

6.6 FreeIPA client configuration

FreeIPA client configuration is done at install time by the FreeIPA client installer. The installer joins the client machine to a FreeIPA domain and configures the client system to use FreeIPA for authorization and authentication [14]. The central component on a FreeIPA client system is SSSD – it provides the authentication and authorization services to the system.

To enable the OpenSSH integration feature, the installer must do additional configuration of SSSD and configure `ssh` and `sshd`. In SSSD configuration, the SSH responder must be enabled. `ssh` must be configured to use the fake proxy command and read host public keys from the `known_hosts` file managed by SSSD. `sshd` must be configured to use the authorized keys command, if the necessary patch is included.

Additionally, the installer should read host public keys from the OpenSSH configuration directory and use the management API to set the public keys in the client's host entry. DNS SSHFP records of the client host should be updated as well. The host management plugin can do this automatically. However, the installer might use credentials of an user not authorized to modify DNS records, so the SSHFP records must be updated some other way. Luckily, this is already solved in the installer: to update A and AAAA records of the client host, it does dynamic DNS update using `nsupdate` [12]. This procedure will be used to update the SSHFP records as well.

Chapter 7

Implementation of the extension

I have implemented the extension according to the design, with some minor additions. The development took place publicly on FreeIPA and SSSD mailing lists. The code was reviewed by the FreeIPA and SSSD development teams and adjusted according to their comments. The code was accepted upstream in both FreeIPA and SSSD and is available in releases starting with FreeIPA 2.1.90.pre2 [4] and SSSD 1.8.0beta1 [9].

7.1 FreeIPA

In FreeIPA, both the backend and frontend were extended, as well as the installer.

7.1.1 Backend

The only change necessary on the backend was inclusion of the additions to the schema and ACIs to the directory server initialization and update data files. No changes were needed in the backend code, although there was a bug in ordering of updates, which prevented the schema to be updated, but it was fixed separately.

The schema was extended with the following definitions:

- attribute type `ipaSsshPubKey` (OID 2.16.840.1.113730.3.8.11.31), which uses the octet string syntax,
- abstract object class `ipaSshGroupOfPubKeys` (OID 2.16.840.1.113730.3.8.12.11), which has one optional attribute `ipaSshPubkey`,
- auxiliary object class `ipaSshUser` (OID 2.16.840.1.113730.3.8.12.12) derived from `ipaSshGroupOfPubkeys`,
- auxiliary object class `ipaSshHost` (OID 2.16.840.1.113730.3.8.12.13) derived from `ipaSshGroupOfPubKeys`.

The set of default access control objects was extended as follows:

- new permission `Manage User SSH Public Keys` was added, allowing write access to `ipaSshPubKey` attributes in user entries,
- the `User Administrators` privilege was modified to include `Manage User SSH Public Keys` permission by default,

- new permission `Manage Host SSH Public Keys` was added, allowing write access to `ipaSshPubKey` attributes in host entries,
- the `Host Administrators` privilege was modified to include `Manage Host SSH Public Keys` permission by default,
- new selfservice `Users can manage their own SSH public keys` was added, allowing users write access to `ipaSshPubKey` attributes in their respective entries.

An appropriate accompanying ACI was defined for each of these objects. Additionaly, host ACIs were extended with ACIs allowing hosts write access to `ipaSshPubKey` attributes in their own entry and in entries of hosts managed by them.

7.1.2 Frontend

The changes on the frontend consist mainly of additions to the management plugins. Two new attributes were added to the `user` and `host` object plugins: `sshpubkey` is the attribute representing the `ipaSshPubKey` LDAP attribute and `sshpubkeyfp` is the virtual attribute used for returning fingerprints of the public keys.

The `sshpubkey` attribute can be set in `user-add`, `user-mod`, `host-add` and `host-mod` commands as a base64-encoded public key blob. The blob is validated to be in the public key format as defined in RFC 4253 [42]. In addition to that, only one public key per public key algorithm is allowed on hosts. The attribute is not included in the output of user and host commands by default, it is included only when it is being modified or when the `all` flag is set. Because the attribute contains binary data, it is not searched in `user-find` and `host-find` commands.

The `sshpubkeyfp` attribute cannot be set, but is included in the output of user and host commands. It contains the fingerprint of each public key plus the public key algorithm name. The fingerprint is returned in the same format OpenSSH uses when displaying public key fingerprints to user: a MD5 fingerprint in hexadecimal with octets separated by colons. The attribute is not searched in `user-find` and `host-find` commands, because that would require generating the fingerprints for each and every user and host.

The `host-add`, `host-mod` and `host-del` commands allow updating DNS SSHFP records. The update is done only when the `updatedns` flag is set. When the DNS service is not installed, the flag is not effective. The flag was missing in the `host-mod` command, so it was added. Before updating the SSHFP records, the correct DNS zone for the host is looked up using the `dnszone-find` command. The SSHFP update itself is done using the `dnsrecord-mod` command.

7.1.3 Installer

In the installer code, most of the changes are in the client install script, `ipa-client-install`, but some modifications were necessary in the server installer code as well. The main additions to `ipa-client-install` are the update of public keys in the client's host entry and configuration of OpenSSH.

Before the public key update is done, the public key are obtained from OpenSSH configuration files. OpenSSH stores host public keys in the global configuration directory (usually `/etc/ssh`) in the files `ssh_host_<algorithm>_key.pub` [34]. These files are read and all valid SSH public keys are extracted from them.

The public key update itself is done in two parts: the first part is calling the host-mod command to update the keys in the client host entry and the second part is executing the nsupdate utility to update DNS SSHFP records. In order to be able to call the host-mod command, the FreeIPA API must be initialized. As ipa-client-install was missing the initialization code, it had to be added. The host-mod command is called with the updatedns flag unset, so that it does not attempt to update the SSHFP records. To update the SSHFP records, nsupdate must be fed a zone update file. The file is generated from the public keys and contains commands to delete all existing SSHFP records under the client's domain name and add new SSHFP records matching the public keys. The SSHFP update can be disabled using new ipa-client-install command-line option --no-dns-sshfp.

The OpenSSH configuration consists of modifications to configuration files of SSSD, sshd and ssh. SSSD configuration is already done in ipa-client-install, so it was not necessary to add new code to handle it. However, it was necessary to add code to activate the SSH service in SSSD configuration. The job of modifiying OpenSSH configuration files is new to ipa-client-install, so a new function had to be added for this purpose.

Changes to the sshd configuration file, sshd_config [35], include setting the AuthorizedKeysCommand option to the path of the authorized keys command provided by SSSD and unsetting the AuthorizedKeysCommandRunAs, so that the command is run as the user running sshd. Because these options are available only in patched sshd, it is checked whether sshd supports them or not before the change to sshd_config is made. There is also an alternative implementation of the patch which uses options named PubKeyAgent and PubKeyAgentRunAs, so they are tried as well. After the changes to sshd_config are done, sshd is restarted to take the new configuration into account. The configuration of sshd can be skipped using new ipa-client-install command-line option --no-sshd.

In the ssh configuration file, ssh_config [33], the ProxyCommand option is set to the path to the fake proxy command provided by SSSD and the GlobalKnownHostsFile2 options is set to the path to the known_hosts file managed by SSSD. As an alternative to this, a new ipa-client-install command-line option --ssh-trust-dns was added. It enables the VerifyHostKeyDNS option in ssh_config, so that host public keys are verified against DNS instead of using SSSD.

The new ipa-client-install command-line options were also added to server install scripts ipa-server-install and ipa-replica-install, which execute ipa-client-install internally. A small additional change to the server installer code was necessary to make the public key update in ipa-client-install actually work: the default dynamic DNS update policy was extended to allow updating SSHFP records.

7.2 SSSD

The modifications in SSSD are more extensive than in FreeIPA. There are changes in the backend and new responder and clients that had to be written from scratch. The build system was extended to allow building SSH-related features conditionally.

7.2.1 Backend

In the SSSD backend, the data provider as well as the LDAP and FreeIPA providers were extended to support retrieval and caching of user and host SSH public keys.

As far as user public keys are concerned, the user attribute map in the LDAP and FreeIPA providers was extended to support them. The name of the LDAP public key attribute is configurable using the `ldap_user_ssh_public_key` configuration option and defaults to `ipaSshPubKey` in the FreeIPA provider. In the LDAP provider, the attribute is disabled by default.

The cache in SSSD uses ldb, a lightweight embedded LDAP-like on-disk database [36]. Attributes in ldb are stored as null-terminated strings. However, SSH public keys are binary blobs, which may contain null characters anywhere in them, so they cannot be stored directly in the cache. In order to rectify this, `ipaSshPubKey` attribute values are base64-encoded in the LDAP code before they are stored in the cache.

To support requesting information about hosts, new handler type, called `hostid`, was created in the data provider. A domain can be configured to use a specific remote service provider to handle host information requests using the `hostid_provider` domain configuration option. By default, the same provider used to provide identity information (configurable using the `id_provider` domain option) is used to provide host information as well.

A `hostid` handler was implemented in the FreeIPA provider. It retrieves host public keys from FreeIPA servers and stores them in the cache. It also allows setting host name aliases for the cached public keys. The FreeIPA provider uses an attribute map to describe the attributes of hosts. The public key attribute was added to the attribute map. The name of the LDAP public key attribute can be configured using the `ipa_host_ssh_public_key` configuration option and defaults to `ipaSshPubKey`.

7.2.2 Responder

The SSSD part of the interface for getting SSH public keys from FreeIPA is implemented in the SSH responder, `sssd_ssh`. The responder dispatches commands for getting public keys of users or hosts from SSSD clients. It is automatically started by SSSD if the `ssh` service is active in SSSD configuration.

The responder listens on UNIX socket `/var/lib/sss/pipes/ssh` for commands from clients. A custom protocol is used for communication between the responder and the clients. Command dispatch begins with parsing of the request received from a client. The configured domains are then searched for the requested entity (user or host). If a domain name is explicitly given in the request, only that specific domain is searched. The search in each domain consists of a data provider request to update the cache entry for the requested entity and a check to see if the entry exists in the cache. If the entry does not exist, next domain is searched. This is repeated until there are no domains to search in and the search returns a failure. If the entry exists, the search is stopped and public keys of the entity are returned. A reply is built from the result and sent back to the client.

Additionaly, the command for getting host public keys allows an alias to be specified for the requested host. If the alias was specified, it is sent along with the fully-qualified domain name of the host in the data provider request.

The managed `known_hosts` file is generated by the responder. The file is located in SSSD public configuration directory, the full path is `/var/lib/sss/pubconf/known_hosts`. The `known_hosts` file is regenerated each time a host public key command is dispatched, after the cache is updated. The file is generated by converting the public keys stored in the cache to the known hosts format [34] and storing the result in the file.

OpenSSH supports hashing of host names in `known_hosts` files [34]. There was a feature request to hash host names in the managed `known_hosts` file as well. This feature was implemented and is enabled by default. It can be disabled by setting the `ssh_hash_known_hosts` ssh configuration option to `false`.

7.2.3 Clients

The interface between SSSD and OpenSSH is formed by the authorized keys command and the fake proxy command. The commands were implemented and are available in SSSD under the names `sss_ssh_authorizedkeys` and `sss_ssh_knownhostsproxy`.

The `sss_ssh_authorizedkeys` command can be executed as follows:

```
sss_ssh_authorizedkeys [-d <domain>] <user>
```

The command requests public keys for the specified user from the SSH responder. The search for the public keys can be restricted to a specific domain. If the request is successful, the command writes the public keys in the authorized keys format [34] to standard output. Errors are reported on the standard error output.

The `sss_ssh_knownhostsproxy` command can be executed as follows:

```
sss_ssh_knownhostsproxy [-d <domain>] [-p <port>] <host> [<proxy command>]
```

The command first attempts to canonicalize the given host name by doing DNS forward and reverse lookups. Then, it requests public keys for the host from the SSH responder using the canonicalized name. The original name is used as an alias in the request. The search for the public keys can be restricted to a specific domain. The result of the request is discarded, as the requested public keys are stored in the `known_hosts` file managed by the SSH responder. Finally, a connection to the host is estabilished and piped to the standard input and output of the command. The connection is estabilished either by opening a socket to the specified host and port, or using the subordinate proxy command, if it is specified. Errors are reported on the standard error output.

Chapter 8

Installation and usage

To demonstrate the features implemented in the extension, the process of building SSSD and FreeIPA from sources, setting up a FreeIPA server and client and usage of SSH features will be described. An example domain example.com will be set up with one server and one client. The operating system used in the demonstration is Fedora 17 Beta x86_64.

8.1 Building from sources

Building FreeIPA and SSSD from sources on Fedora 17 is purely optional, as they are both packaged in Fedora software repositories.

Before building SSSD, all the build dependencies must be installed:

```
[jcholast@fedora17 ~]# yum install autoconf automake bind-utils c-ares-deve
l check-devel dbus-devel dbus-libs docbook-style-xsl doxygen findutils gett
ext-devel keyutils-libs-devel krb5-devel libcollection-devel libdhash-devel
 libini_config-devel libldb-devel libnl-devel libselinux-devel libsemanage-
devel libtalloc-devel libtdb-devel libtevent-devel libtool libunistring-dev
el libxml2 libxslt m4 nspr-devel nss-devel openldap-devel pam-devel pcre-de
vel pkgconfig popt-devel python-devel
```

The SSSD source tarball needs to be downloaded and unpacked:

```
[jcholast@fedora17 ~]$ wget https://fedorahosted.org/released/sssd/sssd-1.8
.3.tar.gz
[jcholast@fedora17 ~]$ tar -xzf sssd-1.8.3.tar.gz
```

Then SSSD RPM packages can be built:

```
[jcholast@fedora17 ~]$ cd sssd-1.8.3
[jcholast@fedora17 sssd-1.8.3]$ autoreconf -fi
[jcholast@fedora17 sssd-1.8.3]$ ./configure
[jcholast@fedora17 sssd-1.8.3]$ make rpms
```

Before building FreeIPA, all the build dependencies must be installed:

```
[root@fedora17 ~]# yum install authconfig autoconf automake 389-ds-base-dev
el gettext krb5-devel krb5-workstation libcurl-devel libtool libuuid-devel
m4 nspr-devel nss-devel openldap-devel openssl-devel policycoreutils popt-d
evel pylint pyOpenSSL python-devel python-kerberos python-krbV python-ldap
python-lxml python-memcached python-netaddr python-nss python-polib python-
pyasn1 python-rhsm python-setuptools selinux-policy-devel svrcore-devel sys
temd-units xmlrpc-c-devel
```

SSSD must be installed too, we can use the packages built in the previous step:

```
[root@fedora17 ~]# yum localinstall sssd-1.8.3/rpmbuild/RPMS/x86_64/{libipa
_hbac,libipa_hbac-python,sssd,sssd-client}-1.8.3-0.fc17.x86_64.rpm
```

The FreeIPA source tarball needs to be downloaded and unpacked:

```
[jcholast@fedora17 ~]$ wget http://freeipa.org/downloads/src/freeipa-2.2.0.
tar.gz
[jcholast@fedora17 ~]$ tar -xzf freeipa-2.2.0.tar.gz
```

Then FreeIPA RPM packages can be built:

```
[jcholast@fedora17 ~]$ cd freeipa-2.2.0
[jcholast@fedora17 ~]$ make rpms
```

8.2 Server setup

The first step of setting up a FreeIPA domain is setting up a FreeIPA server. The server system used in this example is configured to use host name ipaserver.example.com and IP address 192.168.1.100.

The FreeIPA server package must be installed on the system before the setup. The package is available in Fedora and can be installed using `yum`:

```
[root@ipaserver ~]# yum install freeipa-server
```

As an alternative, the packages built in the previous section can be installed instead (including the SSSD packages required by FreeIPA):

```
[root@ipaserver ~]# yum localinstall freeipa-2.2.0/dist/rpms/freeipa-{admin
tools,client,python,server,server-selinux}-2.2.0-0.fc17.x86_64.rpm sssd-1.8
.3/rpmbuild/RPMS/x86_64/{libipa_hbac,libipa_hbac-python,sssd,sssd-client}-1
.8.3-0.fc17.x86_64.rpm
```

BIND and BIND LDAP plugin must be installed as well for DNS server support:

```
[root@ipaserver ~]# yum install bind bind-dyndb-ldap
```

FreeIPA server can now be configured on the system. This is done using the `ipa-server-install` command:

```
[root@ipaserver ~]# ipa-server-install -U -n example.com -r EXAMPLE.COM -p
password -a password --setup-dns --forwarder 192.168.1.1
```

This command will configure FreeIPA server for the example.com domain, using EXAMPLE.COM as the Kerberos realm name and „password“ as the directory server and FreeIPA administrators' password. The DNS service will be configured, using the DNS server originally configured on the system (in `/etc/resolv.conf`) as a forwarder. See the manual page of `ipa-server-install` for description of all the available options.

Finally, synchronization of DNS A and AAAA records with PTR records must be enabled in order for PTR record to be automatically created for FreeIPA clients. To do that, authenticate as admin and enable PTR record synchronization in the example.com DNS zone:

```
[jcholast@ipaserver]$ kinit admin
[jcholast@ipaserver]$ ipa dnszone-mod example.com --allow-sync-ptr=true
```

8.3 Client setup

To be able to use client machines in a FreeIPA domain, they first must be enrolled. In this example, we will enroll only one client, which is enough for demonstration of the OpenSSH integration. The client system is configured to use host name ipaclient.example.com and IP address 192.168.101.

The FreeIPA client package must be installed on the system before the client can be enrolled. The package can be installed using `yum`:

```
[root@ipaclient ~]# yum install freeipa-client
```

Alternatively, the packages built from source can be installed instead:

```
[root@ipaclient ~]# yum localinstall freeipa-2.2.0/dist/rpms/freeipa-{clien
t,python}-2.2.0-0.fc17.x86_64.rpm sssd-1.8.3/rpmbuild/RPMS/x86_64/{libipa_h
bac,libipa_hbac-python,sssd,sssd-client}-1.8.3-0.fc17.x86_64.rpm
```

FreeIPA clients can be enrolled to a FreeIPA domain using the `ipa-client-install` command. Before the command can be executed, the client system must be configured to use the FreeIPA server (192.168.1.100) as DNS server in `/etc/resolv.conf`. Without this, automatic discovery of server setting would not work. After DNS is configured on the system, `ipa-client-install` can be executed:

```
[root@ipaclient ~]# ipa-client-install -U -p admin -w password
```

This command will join the client to the FreeIPA domain example.com using the admin user's credentials. See the manual page of `ipa-client-install` for a detailed description of all the available options.

8.4 Using OpenSSH with FreeIPA

First, we should authenticate as admin:

```
[jcholast@ipaserver ~]$ kinit admin
```

Verify that the server host entry has the correct SSH public keys set:

```
[jcholast@ipaserver ~]$ ipa host-show ipaserver.example.com --all
  Host name: ipaserver.example.com
  Principal name: host/ipaserver.example.com@EXAMPLE.COM
  SSH public key fingerprint: 5A:CE:70:8F:A3:AF:57:C1:D1:C0:C6:28:FC:D4:42:
07 (ssh-dss), 76:2B:1F:98:1C:02:EE:29:43:C1:18:FD:75:57:36:8F (ssh-rsa)
  Password: False
  Keytab: True
  Managed by: ipaserver.example.com
[jcholast@ipaserver ~]$ ssh-keygen -l -f /etc/ssh/ssh_host_dsa_key.pub
1024 5a:ce:70:8f:a3:af:57:c1:d1:c0:c6:28:fc:d4:42:07   (DSA)
[jcholast@ipaserver ~]$ ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key.pub
2048 76:2b:1f:98:1c:02:ee:29:43:c1:18:fd:75:57:36:8f   (RSA)
```

The same procedure can be used to verify host public keys of the client.

Verify that DNS SSHFP records were updated correctly:

```
[jcholast@ipaserver ~]$ dig +short ipaserver.example.com SSHFP
2 1 D017B7B96C1CF0DC9A9CC317AED198EBE61C8369
1 1 EEA71C3819354013613013666B2E4E2627CB470CD
[jcholast@ipaserver ~]$ ssh-keygen -r ipaserver.example.com -f /etc/ssh/ssh
_host_dsa_key.pub
ipaserver.example.com IN SSHFP 2 1 d017b7b96c1cf0dc9a9cc317aed198ebe61c8369
[jcholast@ipaserver ~]$ ssh-keygen -r ipaserver.example.com -f /etc/ssh/ssh
_host_rsa_key.pub
ipaserver.example.com IN SSHFP 1 1 eea71c381935401361301366b2e4e2627cb470cd
```

Again, the same procedure can be used to verify DNS SSHFP records of the client.

Public keys for an user are not automatically updated. Generate a SSH keypair and create new FreeIPA user with the public key set:

```
[jcholast@ipaserver ~]$ ssh-keygen -t rsa
[jcholast@ipaserver ~]$ ipa user-add jcholast --uid=1000 --first=Jan --last
=Cholasta --sshpubkey=‘awk ’{ print $2 }’ .ssh/id_rsa.pub‘
```

Verify that the user entry has the correct SSH public key set:

```
[jcholast@ipaserver ~]$ ipa user-show jcholast
  User login: jcholast
  First name: Jan
  Last name: Cholasta
  Home directory: /home/jcholast
  Login shell: /bin/sh
  UID: 1000
  GID: 1000
  Account disabled: False
  SSH public key fingerprint: 38:FA:5A:79:DF:21:D6:C6:EC:F0:5C:98:8A:4F:AF:
04 (ssh-rsa)
  Password: False
  Member of groups: ipausers
  Kerberos keys available: False
```

```
[jcholast@ipaserver ~]$ ssh-keygen -l -f .ssh/id_rsa.pub
2048 38:fa:5a:79:df:21:d6:c6:ec:f0:5c:98:8a:4f:af:04  jcholast@ipaserver.ex
ample.com (RSA)
```

Now that public keys for both hosts and user are set, we can try using `ssh` to log in remotely from the server to the client and vice-versa:

```
[jcholast@ipaserver ~]$ ssh jcholast@ipaclient
[jcholast@ipaclient ~]$ ssh jcholast@ipaserver
```

Both these commands should work without any warnings or errors and should not prompt for verification of host identity.

Chapter 9

Conclusion

This paper summarizes the state of SSH-related features in FreeIPA and SSSD and describes the design and implementation of an extension providing tighter integration with SSH in these applications. The extension, developed by the author of this paper, has been included in upstream releases of FreeIPA and SSSD, starting with FreeIPA 2.2.0 alpha 2 and SSSD 1.8.0beta1. Packages for these or newer releases of FreeIPA and SSSD will be shipped in Fedora 17, Ubuntu 12.10, future Red Hat Enterprise Linux releases and other Linux-based operating system distributions.

The extension allows user and host SSH public key management in FreeIPA domains and integrates OpenSSH into the FreeIPA ecosystem. However, it implements only the core functionality necessary for proper SSH support in FreeIPA.

One of the areas where the extension is lacking is policy management. Currently users can be allowed or denied write access to their own SSH public keys, SSH public keys of other users or SSH public keys of hosts. It should be made possible to require users to authenticate using their password when modifying their own SSH public keys, similar to how password change works. Administrators should have the ability to force OpenSSH authorized keys options on users or groups of users, such as forcing a specific command to be executed when the user logs in. Impersonation of users should be made possible by allowing users to authenticate using their SSH public keys as other users in a controlled manner.

The extension is not compatible with the OpenSSH-LPK LDAP schema, which several existing applications use. FreeIPA should be extended to provide an OpenSSH-LPK-compatible user subtree on the directory server. SSSD should be extended to allow retrieving user SSH public keys from a generic LDAP server using the OpenSSH-LPK schema.

Bibliography

[1] E. Auge. OpenSSH-LPK Website. http://code.google.com/p/openssh-lpk/. [Online].

[2] E. Auge. OpenSSH-LPK Documentation. http://code.google.com/p/openssh-lpk/wiki/Main, February 2010. [Online; accessed 2012-05-03].

[3] D. J. Barret and R. E. Silverman. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly Media, February 2001. ISBN 0-596-00011-1.

[4] R. Crittenden. FreeIPA 2.1.90 alpha 2 release notes. http://freeipa.org/page/IPAv2_2190_alpha2, February 2012. [Online; accessed 2012-05-10].

[5] F. Cusack and M. Forssen. Generic Message Exchange Authentication for the Secure Shell Protocol (SSH). RFC 4256 (Proposed Standard), January 2006.

[6] J. Galbraith and O. Saarenmaa. The Secure Shell (SSH) File Transfer Protocol (Internet Draft, version 13), January 2007.

[7] J. Galbraith and R. Thayer. The Secure Shell (SSH) Public Key File Format. RFC 4716 (Informational), November 2006.

[8] S. Gallagher. Fedora Features: SSSD By Default. http://fedoraproject.org/wiki/Features/SSSDByDefault, February 2010. [Online; accessed 2011-12-30].

[9] S. Gallagher. SSSD 1.8.0beta1 release notes. https://fedorahosted.org/sssd/wiki/Releases/Notes-1.8.0beta1, February 2012. [Online; accessed 2012-05-10].

[10] J. Hrozek and M. Nagy. FreeIPA and SSSD: Free software identity management. http://rvokal.fedorapeople.org/devconf/freeipa.pp.pdf, September 2009. [Online; accessed 2011-12-30].

[11] J. Hutzelman, J. Salowey, J. Galbraith, and V. Welch. Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol. RFC 4462 (Proposed Standard), May 2006.

[12] Internet Systems Consortium. BIND 9 manual pages: nsupdate(8). http://ftp.isc.org/isc/bind9/cur/9.9/doc/arm/man.nsupdate.html. [Online; accessed 2012-05-12].

[13] Internet Systems Consortium. BIND Website. http://www.isc.org/software/bind. [Online].

[14] E. D. Lackey. FreeIPA: Identity/Policy Management. http://docs.fedoraproject.org/en-US/Fedora/15/html-single/FreeIPA_Guide/index.html, 2011. [Online; accessed 2011-12-30].

[15] J. P. Lanza. Weak CRC allows last block of IDEA-encrypted SSH packet to be changed without notice. http://www.kb.cert.org/vuls/id/315308, January 2001. [Online; accessed 2011-12-30].

[16] J. P. Lanza. Weak CRC allows packet injection into SSH sessions encrypted with block ciphers. http://www.kb.cert.org/vuls/id/13877, November 2001. [Online; accessed 2011-12-30].

[17] J. P. Lanza and S. Van Ittersum. SSH-1 allows client authentication to be forwarded by a malicious server to another server. http://www.kb.cert.org/vuls/id/684820, January 2001. [Online; accessed 2011-12-30].

[18] R. Megginson. 389 Directory Server Features. http://directory.fedoraproject.org/wiki/Features, April 2011. [Online; accessed 2011-12-30].

[19] MIT. Kerberos Website. http://web.mit.edu/kerberos/. [Online].

[20] D. Pal. IPA Client Design Overview. http://www.freeipa.org/page/IPA_Client_Design_Overview, September 2008. [Online; accessed 2011-12-30].

[21] Red Hat, Inc. Red Hat Directory Server 8.2 Administration Guide. http://docs.redhat.com/docs/en-US/Red_Hat_Directory_Server/8.2/pdf/Administration_Guide/Red_Hat_Directory_Server-8.2-Administration_Guide-en-US.pdf, August 2010. [Online; accessed 2012-05-10].

[22] J. Schlyter and W. Griffin. Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints. RFC 4255 (Proposed Standard), January 2006.

[23] D. Stebila and J. Green. Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer. RFC 5656 (Proposed Standard), December 2009.

[24] The 389 Directory Server Team. 389 Directory Server Website. http://directory.fedoraproject.org/. [Online].

[25] The Apache Software Foundation. The Apache HTTP Server Project Website. http://httpd.apache.org. [Online].

[26] The Dogtag Certificate System Team. Dogtag Certificate System Website. http://pki.fedoraproject.org/wiki/PKI_Main_Page. [Online].

[27] The Fedora Project. Fedora Project Website. http://fedoraproject.org/. [Online].

[28] The FreeIPA Team. FreeIPA Website. http://freeipa.org. [Online].

[29] The FreeIPA Team. About FreeIPA. http://freeipa.org/page/About, December 2010. [Online; accessed 2011-12-30].

[30] The OpenBSD Project. OpenBSD Website. http://www.openbsd.org. [Online].

[31] The OpenBSD Project. OpenSSH Website. http://www.openssh.com. [Online].

[32] The OpenBSD Project. OpenSSH Project History and Credits. http://www.openssh.com/history.html, December 2004. [Online; accessed 2011-12-30].

[33] The OpenBSD Project. OpenBSD System Manager's Manual: ssh_config(5). http://www.openbsd.org/cgi-bin/man.cgi?query=ssh_config&sektion=5, April 2012. [Online; accessed 2012-05-10].

[34] The OpenBSD Project. OpenBSD System Manager's Manual: sshd(8). http://www.openbsd.org/cgi-bin/man.cgi?query=sshd&sektion=8, May 2012. [Online; accessed 2012-05-10].

[35] The OpenBSD Project. OpenBSD System Manager's Manual: sshd_config(5). http://www.openbsd.org/cgi-bin/man.cgi?query=sshd_config&sektion=5, May 2012. [Online; accessed 2012-05-10].

[36] The Samba Team. ldb Website. http://ldb.samba.org/. [Online].

[37] The SSSD Team. SSSD Website. https://fedorahosted.org/sssd/. [Online].

[38] T. Ylonen. SSH Change Log. http://www.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/Attic/ChangeLog, November 1995. [Online; accessed 2011-12-30].

[39] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard), January 2006.

[40] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard), January 2006.

[41] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[42] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), January 2006.