HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Computer Science Degree Program

ENABLING MULTIPLE HIP IDENTITIES ON LINUX

Master's Thesis

Niklas Karlsson

Telecommunications Software and Multimedia Laboratory
Espoo 2005

HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Computer Science Degree Program

ABSTRACT OF MASTER'S THESIS

Author: Niklas Karlsson

Title of thesis:
Enabling Multiple HIP Identities on Linux

Date: 8th August 2005 **Pages:** 11 + 83
Professorship: Telecommunications Software **Code:** T-109
Supervisor: Professor Antti Ylä-Jääski
Instructor: Janne Lindqvist, M.Sc. (Tech.)

The Host Identity Protocol (HIP) is an approach to revise the double role of IP addresses by separating endpoint identification from topological localization. HIP introduces a new cryptographic Host Identity name space. In HIP architecture, a host accommodates one or more Host Identifiers (HI) that identify distinct HIP endpoints.

The goal of this thesis was to design and implement support for multiple HIP identities per host. The work was conducted by extending the HIP for Linux (HIPL) implementation in the Linux kernel. A prototype implementation was developed by enhancing various components of HIPL. The kernel component was modified to support multiple identifiers. In addition, endpoint resolution in user space was enabled for a multitude of HIs.

The outcome of the implementation and possible implications were analyzed. Additionally, some potential targets for future work and further research were identified. The implementation works as an enabler for user-specific HIP identifiers. Additionally, the support for multiple endpoints within a host serves as a basis for more advanced concepts such as application mobility.

Keywords: HIP, HI, Linux, multiple, identity, identifier
Language: English

TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Niklas Karlsson	
Työn nimi:	Usean HIP-identiteetin tuki Linuxille	
Päiväys:	8. elokuuta 2005	**Sivumäärä:** 11 + 83
Professuuri:	Tietoliikenneohjelmistot	**Koodi:** T-109
Työn valvoja:	Professori Antti Ylä-Jääski	
Työn ohjaaja:	DI Janne Lindqvist	

HIP (Host Identity Protocol) on uudenlainen tietoliikenneprotokolla, joka pyrkii parantamaan TCP/IP-protokollapinon arkkitehtuuria. Nykyisessä arkkitehtuurissa IP-osoite sekä identifioi kommunikoivan päätelaitteen että kertoo sen reititykseen liittyvän topologisen sijainnin. HIP-protokolla muuttaa vanhaa arkkitehtuuria hoitamalla verkkolaitteiden identifikaation uuden kryptografisen nimiavaruuden avulla. Jokaista verkkoon liitettyä päätelaitetta vastaa yksi tai useampi tunniste (Host Identifier), joka identifioi laitteen yksiselitteisesti.

Tämän diplomityön tavoitteena oli suunnitella ja toteuttaa tuki useammalle HIP-identiteetille. Työ tehtiin jatkokehittämällä olemassaolevaa HIPL (HIP for Linux) -implementaatiota. Työn tuloksena syntyi Linux-käyttöjärjestelmän ytimessä toimiva prototyyppitoteutus. Linux-ytimessä toimivan HIPL-moduulin muokkaamisen lisäksi myös käyttäjäavaruuden osoitteenmuunnostoiminnallisuutta (Resolver) parannettiin useita tunnisteita tukevaksi.

Työn tuloksena syntynyt toteutus analysoitiin. Lisäksi esiteltiin työn kehittämisen yhteydessä syntyneet ideat jatkokehitystä ja -tutkimusta silmälläpitäen. Toteutus mahdollistaa käyttäjäkohtaiset HIP-tunnisteet. Samalla se toimii pohjana lisätutkimukselle kuten liikkuville sovelluksille (Application Mobility).

Avainsanat:	HIP, HI, Linux, identiteetti, tunniste
Kieli:	Englanti

Acknowledgments

This thesis was written for InfraHIP project in Espoo, Otaniemi. I wish to express my gratitude to Telecommunications Software and Multimedia Laboratory for making this work possible.

I would like to thank my Professor, Antti Ylä-Jääski, for supervising my thesis and for giving me the opportunity to do this work. I also want to thank my instructor, Janne Lindqvist, for giving me valuable comments and for correcting the spelling and outlook.

Especially, I want to thank Miika Komu for his time and help with the practical issues concerning the HIPL implementation. Miika also gave me many invaluable comments regarding the contents of my thesis.

Finally, I would like to thank all helpful people, my girlfriend, my family and friends for their support and encouragement

Espoo August 8th 2005

Niklas Karlsson

Abbreviations and Acronyms

API	Application Programming Interface
DNS	Domain Name System
DoS	Denial of Service
DSA	Digital Signature Algorithm
ED	Endpoint Descriptor
EID	Endpoint Identifier
ESP	Encapsulating Security Payload
FQDN	Fully Qualified Domain Name
GID	Group Identifier
HI	Host Identifier
HIP	Host Identity Protocol
HIPL	Host Identity Protocol for Linux
HIT	Host Identity Tag
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPsec	Internet Protocol security
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
LSI	Local Scope Identifier
MAC	Media Access Control
PEM	Privacy Enhanced Mail
PID	Process Identifier
RR	Resource Record
RSA	Rivest Shamir Adleman public key algorithm

SA	Security Association
SID	Service Identifier
SP	Security Policy
SPI	Security Parameters Index
TCP	Transport Control Protocol
UDP	User Datagram Protocol
UID	User Identifier
ULD	User-level Descriptor

Contents

Abbreviations and Acronyms **v**

1 Introduction **1**

1.1 Scope 2

2 Background **3**

2.1 Related Terminology 3

2.2 Host Identity Protocol 3

2.2.1 Host Identity Namespace 3

2.2.2 Security Considerations 4

2.2.3 Base Exchange 5

2.2.4 HIP - IPsec Integration 6

2.2.5 Other Types of HIP Messages 7

2.3 Layered Naming and Host Identity Protocol 7

3 Host Identity Protocol for Linux **9**

3.1 HIPL Implementation in Linux Kernel 9

3.1.1 Host Identifiers 10

3.1.2 Endpoint Resolution in HIPL 10

3.2 HIPL Related APIs 11

3.2.1 BSD Socket API 12

3.2.2 Legacy HIP API 13

3.2.3 Native HIP API 14

3.3 HIPL Architecture 17
3.3.1 Kernel Based Version 17
3.3.2 User space Version 23

4 Requirements and Evaluation Criteria 25
4.1 Functional Requirements 25
4.1.1 Key Selection 26
4.1.2 HIP Associations 26
4.1.3 Endpoint Resolution 26
4.1.4 Access Control 27
4.1.5 User-specific Identifiers 27
4.2 Non-Functional Requirements 27
4.2.1 Compatibility and Usability 27
4.2.2 Quality 28
4.3 Evaluation Criteria 28

5 Implementation 29
5.1 Data Structures 29
5.1.1 Host Association Database 29
5.1.2 Endpoint Descriptor Database 30
5.1.3 Socket 31
5.2 R1 pre-creation 31
5.3 Bind 32
5.3.1 Using the Bind Routine 33
5.4 Hooks 34
5.5 Resolver 35
5.5.1 Mappings 36
5.6 Endpoint Descriptors 37
5.6.1 Local Endpoints 37
5.6.2 Endpoint Descriptor Life Cycle 39
5.6.3 ED Based Access Control 40

5.7 Hipconf 43
5.8 Public and Anonymous Identifiers 43

6 Analysis **44**
6.1 Functional Analysis 44
6.1.1 Test Environment 44
6.1.2 Multiple Identifiers 45
6.1.3 Multiple HIP Associations 45
6.1.4 Resolver 46
6.1.5 Controlling Access to Application Specified Identifiers . 46
6.1.6 Evaluation Against the Criteria 47
6.2 Consequences of Hosting Multiple HIs 47
6.3 Security Implications 48
6.3.1 Local Point of View 48
6.3.2 Network Point of View 49
6.4 Peer HIT to IPv6 Address Mappings 50
6.4.1 Alternative Resolver Model 50
6.4.2 Separating the Host Associations from Mappings . . . 51

7 Future Work **52**
7.1 HIP Policies 52
7.2 Missing Native API Functionalities 52
7.3 Opportunistic Mode 53
7.4 Loopback 54
7.5 User-specific Identifiers 55
7.6 Pre-assigned System Identifiers 55
7.7 Discarding HIP Associations 55
7.8 Application Mobility 56
7.9 Enforcing Explicit Bind 56
7.10 User space version of HIPL 57

8 Conclusion **58**

A Functional Tests **62**

A.1 Responder Test 62

A.2 Multiple Identifiers and HIP Associations Test 66

A.3 Test Server 70

A.4 Resolver Tests 74

A.4.1 getendpointinfo 74

A.4.2 getaddrinfo 76

A.5 HI Binding Test 80

List of Figures

2.1 HIP base exchange 6

3.1 Endpoint resolution 11
3.2 HIP layering model [9] 12
3.3 Relationships between sockets, EDs and locators [9] 16
3.4 HIPL kernel space version architecture 18
3.5 HIPL hooks inside the Linux networking stack 20
3.6 HIPL user space version architecture 23

5.1 Multiple simultaneous associations 30
5.2 Binding a local HI to a server socket 33
5.3 Binding a local HI to a client socket 34
5.4 HIT-to-IPv6-address mapping with multiple local HIs 37
5.5 Binding to a system HI using the native HIP API 38
5.6 Binding to a system HI using the legacy HIP API 38
5.7 Life cycle of an application specified identifier and the correlated ED 41
5.8 Acces control sequence 42

6.1 Alternative Resolver Model 51

Chapter 1

Introduction

In the early 70's when TCP/IP architecture was first proposed, computers were connected to the network statically. Every network node could be identified by its unchangeable IP address since only one physical network interface was commonly used. A limited number of mostly research-oriented people with very similar motivations used the network. In consequence, the early Internet was a trustworthy environment where the network endpoints and different human stakeholders could trust each other to behave as desired.

At the present, the situation is drastically different. Several new requirements are emerging as the Internet is used for a wider set of purposes than before. A multitude of different individuals and organizations with varying objectives and goals use the worldwide network. Consequently, some completely new demands and pressures on the network infrastructure have appeared. Misuse of the Internet has increased dramatically rendering the network insecure. At the same time, network nodes are turning more and more mobile wanting to change their attachment point to the network dynamically. In spite of these new emerging requirements, the architecture and fundamental characteristics of the Internet have remained rather simple and unchanged.

The traditional IP address suffers from semantic overloading by representing both the topological routing location and also the identifier for a network interface. The shortcomings of the original IP architecture have been realized among researchers along with the development of various mobility and multihoming technologies. Various proposals have been made to separate the identification and location[1, 3].

The Host Identity Protocol (HIP) [16, 13, 14] is a proposal to enhance the Internet protocol suite to better match the current needs. The main idea is to split the transport layer identifiers from the network layer identifiers. A

new protocol layer, HIP, is introduced between the two layers. In this new architecture the transport layer identifiers are location independent HIP endpoints and the network layer identifiers are used solely for routing purposes. The new layering scheme is beneficial from the point of view of mobility since transport layer connections can persist even if the underlying network layer addresses change.

The topic of this thesis suggests that we can realize a networked computing node as a host for multiple distinct communication endpoints. However, the endpoints are not identified by the IP address and port number pairs like before but instead a new level of names is used. A Host Identifier (HI) represents a communication endpoint in the new HIP namespace. In practice, the ability to have multiple HIP layer endpoints on a single host means that there exists a multitude of Host Identifiers in the level of the host machine's operating system.

1.1 Scope

The purpose of this thesis is to design and implement support for multiple HIP identities per host. The actual work comprises of a proof-of-concept implementation done by extending and enhancing the HIP for Linux (HIPL) implementation in the Linux kernel. The implementation includes improvements to the user space resolver library and to the HIP kernel module which are part of the HIPL implementation.

The possibility to accommodate multiple distinct Host Identifiers in a single host raises question about access control. Design and implementation of access control mechanisms for multiple coexisting HIs are included in the scope of this thesis.

Architecturally complex issues like distribution, creation or lifetime of the Host Identifiers are out of scope. Privacy aspects concerning, for example, the anonymous identifiers are also out of scope.

Chapter 2

Background

In this chapter, some important terms that recur in the text are first defined. Then, the Host Identity Protocol and some related topics are discussed.

2.1 Related Terminology

An *endpoint* is one participant of an end-to-end communication process. For example, an endpoint can be a computer node or a network application [18].

The terms *End-host* and *host* refer to a computational system that can host one or many communication endpoints. A host can accommodate a number of simultaneous communication processes that are bound to different endpoints.

Identity refers to an entity or subject that is identified. An identity is a distinct quality that differentiates the corresponding entity umbambiguously from its peers. An identity can refer to a user, to a computing node or to a user group. An *identifier* represents an identity and uniquely identifies it [18].

2.2 Host Identity Protocol

2.2.1 Host Identity Namespace

HIP adds a new namespace, Host Identity, to the current Internet network model. HIP introduces location independent identification on top of the

IP layer. Networking nodes are bound to Host Identities represented by cryptographic identifiers. In this new scheme IP addresses serve as routing locators and HIP layer identifiers differentiate the communication endpoints.

The splitting of the IP layer locators from the upper layer identifiers is an approach to handle mobility and multihoming [18]. HIP revises Transport Layer Identifiers (TLI) by replacing IP addresses with Host Identifiers. This allows transport layer connections to survive even when a mobile node moves from a network attachment point to another.

A Host Identity is represented by a Host Identifier (HI). The HI is the public key of an asymmetric key pair. In addition to the full-length key, the HI can be represented in a more compact form also. A Host Identity Tag (HIT) is a 128-bit hash of the corresponding HI. A HI can have varying size since there are various public key algorithms with varying key lengths. Therefore, the fixed size HIT is used in HIP payloads and to index the protocol state in the end hosts. A third representation form of a HI is the Local Scope Identifier (LSI). LSIs are defined to be either 32-bit or 128-bit so that they are compatible both with the IPv4 and IPv6 APIs. The purpose of the LSI is to provide a compressed representation format that can be used locally.

Host Identifiers can be either public or anonymous. All hosts should have at least two HIs, one reserved for anonymous usage and one public one [13]. HIs can be published by storing them in the Domain Name System (DNS) [15]. Anonymous HIs are typically temporary with a very limited lifetime.

2.2.2 Security Considerations

Currently, Internet hosts are identified by their IP addresses. This has some implications on the security. For instance, it is relatively easy to forge IP packets so that they appear to be originating from a trusted source. Additionally, most of the Internet traffic is still communicated in an unencrypted form. Common IP packets are prone to eavesdropping and message integrity cannot be guaranteed.

HIP introduces strong identification based on cryptographic Host Identifiers. HIP enabled hosts are able to prove their identity by using public key cryptography. A host can authenticate itself by holding the private key that corresponds to a Host Identifier, the public key part of the asymmetric key pair. Furthermore, HIP provides communication confidentiality and message integrity by integrating with the IP Encapsulating Security Payload (ESP) protocol. A separate Internet Draft describes how these two protocols can be utilized together [6].

The initialization part of the Host Identity Protocol has been designed to protect the responder from Denial-of-Service (DoS) attacks. In addition, HIP is designed to be resilient against Man-in-the-Middle (MitM) attacks. However, the protocol is potentially vulnerable to the same attacks in some specific circumstances [14]. For instance, HIP allows to initiate the protocol key exchange without a prior knowledge about the responder's identity. This so called opportunistic mode makes the initiator vulnerable to Man-in-the-Middle attacks.

In the HIP architecture, the Host Identifiers can either be public or unpublished. Communication and location privacy can be provided by using the unpublished HIs since they cannot be queried from a global name service like DNS. For comparison, the current Internet addressing cannot provide anonymity in a consistent way because the global IP addresses are registered and publicly available through the DNS.

HIP architecture specification describes HIP policies that would allow initiators to use a different HI for different responders [13]. A HIP policy restricts the use of local HIs depending on the peer HIT. For instance, only temporary and unpublished HIs could be allowed when initiating the protocol with a certain peer.

2.2.3 Base Exchange

HIP is a key-exchange protocol that has been specifically designed from the point of view of host mobility. The HIP base exchange is a four-packet cryptographic protocol that establishes a state, a HIP association, between two endpoints. Its purpose is to identify the endpoints to each other. Both endpoints are assured that their peer possesses the private key that corresponds to the peer's public Host Identifier. During the base exchange, the endpoints perform Diffie-Hellman (DH) key exchange to create a shared secret. The base exchange results in a pair of IPsec Encapsulated Security Payload (ESP) Security Associations (SAs), one in each direction. The four-way base exchange is illustrated in Figure 2.1.

The HIP base exchange creates a state between an Initiator and a Responder. The Initiator begins the sequence by sending **I1** packet to the Responder. The first packet contains the HITs of both the Initiator and the Responder. The HIP base specification [14] also defines an opportunistic mode. If the Initiator does not know the Responder's identifier, it can still try the opportunistic mode and use a NULL value as the Responder's HIT.

The second packet, **R1**, starts an authenticated Diffie-Hellman key exchange

for session key generation. The packet contains a cryptographic challenge that the Initiator must solve before continuing the handshake. The puzzle complicates Denial-of-Service attacks against the Responder by forcing the Initiator to use some computing cycles for calculations. Furthermore, the R1 message contains a signature and initial Diffie-Hellman parameters. To make HIP base exchange less vulnerable to DoS attacks, R1 messages can be pre-created. In addition, the Responder also remains stateless until it receives I2 message.

The third packet is **I2** and it must contain a solution for the challenge received in R1. Further, the packet contains a Diffie-Hellman parameter and a signature. The base exchange is finished by a signed **R2** packet.

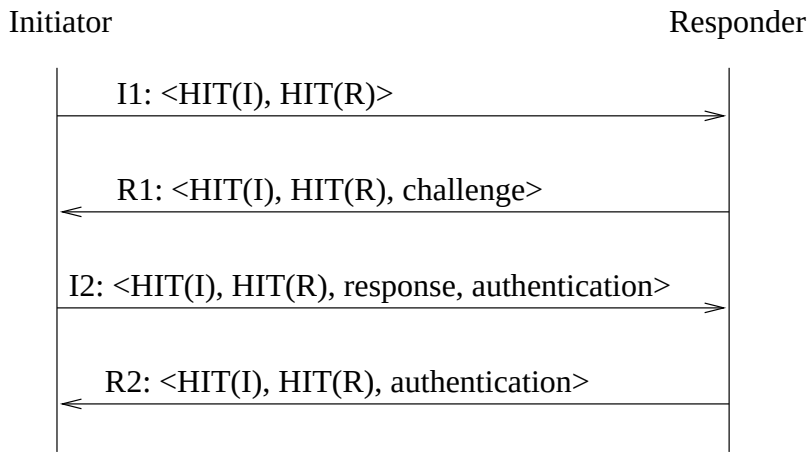

Figure 2.1: HIP base exchange

2.2.4 HIP - IPsec Integration

The HIP architecture specification [13] prefers IPsec for secure exchange of the actual data after the HIP base exchange. An Internet Draft has been published about using ESP transport with HIP [6]. The specification does not restrict the data transfer method to any specific protocol, however.

The HIP base exchange is used to set up a pair of ESP Security Associations between two hosts. The ESP SAs are bound to end-host HITs instead of IP addresses. However, the data packet headers do not include HIs or HITs. Instead, the ESP Security Parameter Index (SPI) values are used to find the right Security Association for any received ESP packet. The SPI values are set up during the HIP base exchange. In a sense, a SPI provides a compressed

representation of a pair of HITs [6]. When an ESP packet arrives to a host, the IP addresses in the packet header are replaced with the corresponding HITs before ESP processing. Similarly, HITs are changed to IP addresses in the outgoing packets.

Each HIP association is associated with two ESP SAs, one for the incoming packets and one for the outgoing ones. A HIP association is closely linked to the related ESP SAs and when it ceases to exist also the corresponding SAs must be removed.

2.2.5 Other Types of HIP Messages

In addition to four message types used in the base exchange, HIP includes other types of messages. The current Internet Draft [14] defines four additional types of HIP packets. One is for updating HIP associations, one is for sending notifications, and two are for closing a HIP association.

Dynamic readdressing is possible by using HIP, as the IP address no longer functions as an endpoint identifier. A mobile node causes changes to the bindings between HIs and IP addresses whenever it moves. **UPDATE** messages are used to renew a HIP association between two hosts. For example, the UPDATE messages can be used for rekeying expiring security associations, adding new associations and for changing IP addresses associated with hosts.

HIP associations are closed using **CLOSE** and **CLOSE_ACK** messages. When a host receives a CLOSE message, it responds with a CLOSE_ACK. After the closing procedure has started, any new data transfer attempts between the same endpoints will trigger a new base exchange that creates a fresh association.

NOTIFY messages are used for passing informational data such as indications about possible error conditions.

2.3 Layered Naming and Host Identity Protocol

The Layered Naming Architecture proposal [1] argues that there should be three levels of name resolution in the current Internet instead of the one currently used, DNS. These resolutions are (1) from user-level descriptors to service identifiers, (2) from service identifiers to endpoint identifiers and (3)

from endpoint identifiers to IP addresses.

There are also three different layers of names on top of the IP addresses: user-level identifiers, service identifiers and endpoint identifiers. User-level descriptors represent the highest layer of naming. ULDs are used in the computer-human interface like in a query string for a search engine. An identifier representing the second highest level of naming, services, is returned as a result to a search. Service identifiers (SIDs) allow applications to refer to services with persistent SID names so that the names are not tied to the endpoints hosting the services anymore.

Applications use another resolution layer to resolve SIDs to endpoint identifiers (EIDs). A session protocol like HTTP can translate a SID to an EID by using the SID resolution service. The meaning of the EID is to uniquely identify a host. The lowest layer of resolution in this scheme is the EID resolution. This resolution layer translates the destination EID to one or more locators, that is, IP addresses.

The HIP protocol represents the lowest level of naming in this arhitecture. HIP [13, 14] provides the EID layer of naming and also the EID resolution functionality. The ability to have multiple HIP identities (EIDs) per host allows several entities sharing the host to be logically separated. It could be feasible that some upper layer names would represent entities like an application, a single user or a user group. Each of these upper layer identities could be bound to a different lower layer HIP identity.

Chapter 3

Host Identity Protocol for Linux

This chapter describes the Host Identity Protocol for Linux (HIPL) implementation. Discussion about the related Application Programming Interfaces (API) is included. Finally, the implementation architecture and the most relevant software components are introduced.

3.1 HIPL Implementation in Linux Kernel

HIP for Linux (HIPL) is a software project at Helsinki Institute for Information Technology (HIIT) and Helsinki University of Technology (TKK). The purpose of the project is to implement the Host Identity Protocol for Linux. The ultimate goal is to get the implementation approved into the official Linux kernel.

The current HIPL implementation is based on the standard Linux kernel. For now, only IPv6 protocol is supported but also IPv4 support is going to be implemented in the future. Currently the implementation works with the kernel version 2.6.11.

The latest publicly released version of the HIPL implementation is kernel based. The implementation consists of the HIP kernel module, HIP socket handler and a user space resolver library. Additionally a simple user space program called *hipconf* is used to create and configure the cryptographic host identities.

The kernel based implementation was not accepted by the Linux kernel community as such. The main reason was the relatively large size of the HIPL kernel module in source code lines. In general, the community is somewhat

reluctant to approve anything new into the kernel if the same functionality could be implemented in the user space. Consequently, the emphasis in the HIPL project has been turned on a new user space version. Both versions of the HIPL are described later in the text and the most significant differences are discussed.

3.1.1 Host Identifiers

In HIPL, Host Identifiers are classified into two different categories: identifiers provided by the operating system and the application specified identifiers. The system specified identifiers are located physically in the */etc/hip/* directory. Advanced HIP aware applications are allowed to specify their own identifiers. Any particular location for the application specified identifiers has not been defined but a natural place would be under the user home directories.

Asymmetric keys are stored in Privacy Enhanced Mail (PEM) formatted files on the local disk. For the moment, HIPL supports RSA and DSA keys. A local HI is communicated to the HIP kernel module by loading the corresponding PEM-encoded private key from the file system. The HI is specifically the public key, which is deduced from the private key inside the kernel.

3.1.2 Endpoint Resolution in HIPL

Network applications commonly use the Fully Qualified Domain Name (FQDN) to establish a connection to a distant peer. A query to the Domain Name System (DNS) is needed to find out the corresponding IP address. The interface that handles the name resolution is known as the *resolver*. The resolver includes a set of functions that are used by applications to translate network addresses to DNS host names and vice versa.

HIP connections use HITs as the endpoint identifiers in the place of the IP addresses. Host name query has to result in HITs instead of IP addresses. Additionally, in order to transmit packets to the right destination, the kernel has to know the IP address that corresponds to the resolved HIT. A mapping from a HIT to an IP address has to be created as a part of the name resolution process.

The endpoint resolution mechanism used in HIPL is visualized in the Figure 3.1. Applications call resolver functions by giving the host name as an

argument. The resolver queries DNS that responds with a HIT and the corresponding IP addresses. Next, the resolver passes the HIT and the locators to the HIPL kernel module. An Endpoint Descriptor (ED) is returned from the kernel module to the resolver in the case the HIP native API is used (see 3.2.3). Lastly, the resolver ends the sequence by returning an Endpoint Descriptor or a HIT to the application. An ED relates to the HIP native API and a HIT refers to the usage of the legacy API (see 3.2.2).

It is notable that in the current HIPL implementation the endpoint resolution is handled with a manually configured */etc/hip/hosts* file since HIP enabled DNS is not available yet. However, an Internet-Draft specifies how the DNS could be utilized with HIP by introducing two new resource records (RRs) [15].

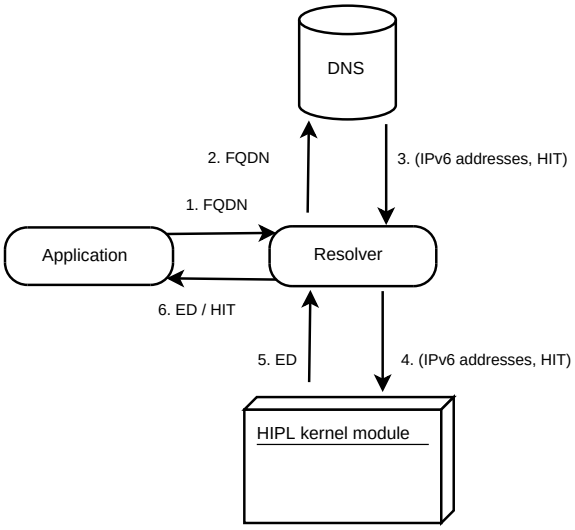

Figure 3.1: Endpoint resolution

3.2 HIPL Related APIs

HIP related Application Programming Interfaces enable HIP for user space applications. The APIs support basic networking functionality needed by the applications, such as identity resolution and binding to sockets. The HIP layering model with the APIs is visualized in the Figure 3.2.

From the point of view of HIP protocol, there are two kinds of network applications. HIP aware applications make use of the HIP protocol explicitly

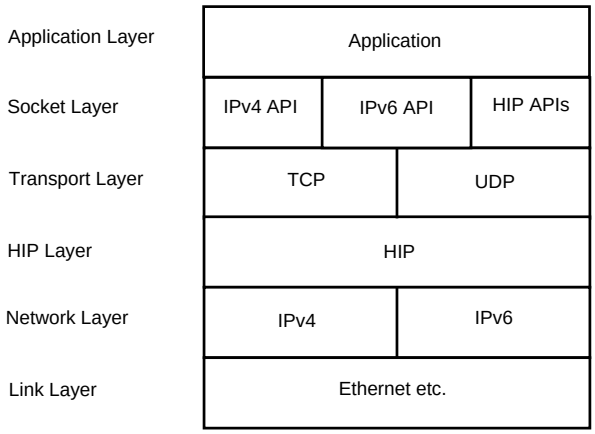

Figure 3.2: HIP layering model [9]

via the HIP native API. Legacy applications use HIP transparently via so called legacy HIP API. In the latter case, the amount of changes in the application code is minimized and applications make use of HIP through the common socket API.

3.2.1 BSD Socket API

The BSD socket API is used for virtually all network communication in UNIX based operating systems and it is therefore also closely related to using HIP. In the following, the most important parts of the API concerning the HIPL implementation and the HIP APIs are briefly described.

In Linux the TCP/IP stack resides in the kernel. All the networking functions performed above the transport layer are done in the user space [5]. Applications make use of the TCP/IP protocols through the socket interface. The socket API describes a set of C language functions for network communication. The API introduces an abstraction known as the *socket*. Regarding network communication, sockets represent communication endpoints.

Applications request the operating system to create communication sockets that are assigned with integer numbers for later referencing. Sockets are created initially without any association to local or destination addresses. An IP address and protocol port number pair is passed to the socket interface using the *sockaddr* structure.

The resolver function *getaddrinfo* is important concerning this thesis. It provides hostname-to-address translation for applications. In other words,

the function accepts a node name as an input argument and returns all addresses that are supported for this name and the specified parameters. Getaddrinfo creates either IPv4 or IPv6 address structures which can be used for bind or connect function calls.

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Server applications use the *bind* command to associate a port and address combination with a socket. *Listen* is used by a server application to indicate that it is ready to receive TCP connections. The function is used to set a queue limit for pending connection requests. *Accept* routine is called by a server when it is ready to accept a connection request. A new socket is returned for the accepted connection.

Clients use the socket call *connect* to when they attempt to initiate a connection on a server socket. When the stream-oriented TCP protocol is used, the command triggers TCP connection negotiation between the communicating peers. When a connectionless protocol such as UDP is used, the connect routine specifies the peer address association for the socket. In general, sockets that are associated with a connection-based protocol may successfully connect only once. Related to the connectionless sockets, the connect routine may be utilized multiple times to change the association.

The socket API functions *send*, *sendto*, *sendmsg*, *recv*, *recvfrom* and *recvmsg* are used for the actual data transmission between two sockets. Send and recv are used with a connected stream-oriented socket. Other calls are for connectionless or UDP sockets.

3.2.2 Legacy HIP API

UNIX based operating systems use a standardized C language library called libc which provides facilities for common operations such as the socket interface. The USAGI project [17] has another implementation of the socket API called libinet6. The latter one is used in the HIPL project and it forms the basis for the modified resolver functions.

From the point of view of this thesis, the legacy HIP API refers to the API functions in the modified libinet6 resolver library. With the legacy API, applications can use the HIP protocol transparently without a need to modify application code. The resolver library is a part of HIPL implementation and

it is described in a publication by Candolin et al [2]. In practice, only the socket API function *getaddrinfo* has been modified to enable the legacy API functionality.

Legacy applications make use of HIP through unmodified socket interface. The 128-bit HITs are compatible with the IPv6 API since also IPv6 addresses have the same size. Consequently, HITs can be used with IPv6 sockets. Socket address structure for the IPv6 sockets is *sockaddr_in6* [4].

The legacy API can be enabled by using two different approaches. If a compile time *transparent HIP mode* option is enabled, getaddrinfo always returns a HIT before IPv6 addresses. The scheme allows falling back to IPv6 addresses. If the resolver cannot find a HIT that matches the given host name, only the IPv6 addresses are returned to the caller. One benefit of the transparent mode is the backwards compatibility with the legacy network applications. Changes to the legacy application code are not required. All that is needed to be done is to ensure that the applications are properly linked to the modified libinet6 resolver library.

In addition to the transparent HIP mode, applications can use HIP via the legacy API explicitly. A special *AI_HIP* flag can be passed to the getaddrinfo function which returns only HITs when the flag is set. However, the explicit mode requires a small change in the application code and therefore forces to recompile.

The endpoint resolution via the legacy API is enabled only for client applications which call the resolver to get an address of a peer host. Local HIP endpoints cannot be resolved to HITs. In consequence, server applications bind to sockets by using the *in6addr_any* constant which denotes that the system is let to select the source address. This means that, in practice, servers accept connections destined for any given local HI.

3.2.3 Native HIP API

An API for HIP aware applications has been proposed in the form of an Internet Draft: Native Application Programming Interfaces for the Host Identity Protocol [9]. The native HIP API forms a part of the HIPL implementation.

The applications that use the native API are HIP aware which means that they have some degree of control over the HIP layer and identifiers. For instance, applications can specify their own Host Identifiers. The API introduces a new protocol family PF_HIP for the socket API. The PF_HIP constant must be passed to the socket function in order to make use of the

Layer	Identifier
Application Layer	ED, port, protocol
Transport Layer	HI, port
HIP Layer	HI
Internet Layer	IP address
Network Access Layer	MAC address

Table 3.1: The modified TCP/IP layering model with EDs

native API.

The Endpoint Descriptor (ED) abstraction is a central concept in the HIP native API. An ED serves as a reference to a Host Identifier. The native API refines the normal HIP namespace model. The HIP layer is situated between the network and transport layers like before. Application layer is refined so that it uses source ED, destination ED, source port, destination port and transport protocol type to distinguish between different connection associations. Table 3.1 illustrates the modified layering model.

Regarding the native API, the interface to the ED abstraction is the *sockaddr_ed* structure. The *ed_family* field is always set to PF_HIP to indicate a native HIP family socket. The *ed_port* denotes a transport layer port number. The *ed_val* is a unique identifier that identifies the ED. It is maintained by the HIP kernel module and used inside the kernel to reference EDs. The data structure definition is shown below.

```
struct sockaddr_ed {
         unsigned short int ed_family;
         in_port_t ed_port;
         sa_ed_t ed_val;
}
```

The native HIP API does not introduce any major changes to the syntax of the common socket API functions. In practice, it differs from the socket API by using sockaddr_ed structures in the place of the common sockaddr_in or sockaddr_in6 structures.

Native HIP Sockets

The relationship between a socket and an ED is many-to-one. Consequently, the same ED can be reused for multiple sockets. Also the relationship be-

tween an ED and a HI is many-to-one. Any given HI may be referred by multiple EDs. The relationships are visualized in the Figure 3.3.

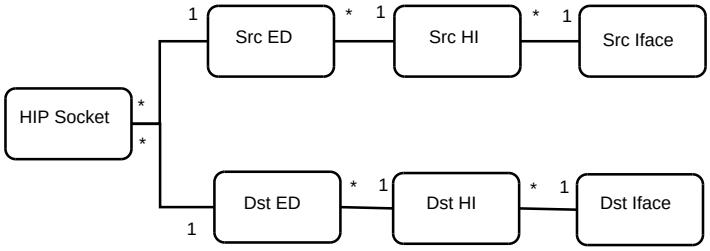

Figure 3.3: Relationships between sockets, EDs and locators [9]

Endpoint Resolution in the Native API

Applications must be able to resolve host names to ED structures in order to use the native API socket calls. The sockaddr_ed structures are resolved by the native API function *getendpointinfo*. The function accepts *nodename*, *servname* and *hints* as input arguments. The result is placed into *res* which is a pointer to a dynamically allocated linked list. Getendpointinfo returns zero on success and a non-zero value to indicate an error condition.

```
int getendpointinfo(const char *nodename,
                    const char *servname,
                    const struct endpointinfo *hints,
                    struct endpointinfo **res);
```

The getendpointinfo resolver function introduces a new datastructure, *endpointinfo*. The structure is used both as the input and output argument for the getendpointinfo. The endpointinfo structure resembles the addrinfo that is used in the socket API. The *ei_endpoint* member points to an address structure that can be used in the native socket API calls.

```
struct endpointinfo {
              int ei_flags;
              int ei_family;
              int ei_socktype;
              int ei_protocol;
              size_t ei_endpoint_len;
```

```
            struct sockaddr *ei_endpoint;
            char *ei_canonname;
            struct endpointinfo *ei_next;
}
```

Application Specified Identities

Application specified endpoints can be loaded by using the native API routine *hip_endpoint_pem_load*. The key files must be PEM encoded. The loaded key is stored in the *endpoint* structure [9].

```
int hip_endpoint_pem_load(const char *filename,
                          struct endpoint **endpoint);
```

The endpoint structure must be converted to an ED in order to use it with native HIP sockets. The API defines two functions for doing the conversion. *getlocaled* converts the local endpoints to *sockaddr_ed* structures and the *getpeered* routine is used in similar way for the peer endpoints.

3.3 HIPL Architecture

This section describes the architecture of the HIPL implementation. Both the kernel space and the user space versions are included in the discussion. The emphasis is on the kernel based implementation since it represents the original work.

3.3.1 Kernel Based Version

The kernel based version of HIPL is visualized in the Figure 3.4. Applications make use of HIP by using the modified libinet6 resolver library. The resolver communicates with the HIPL kernel module. Socket API calls to the native HIP sockets are forwarded to the HIP socket handler in the kernel module. Legacy applications use the ordinary IPv6 socket API but HITs are used in the place of IPv6 addresses. The most important data structures in the kernel are also included in the picture.

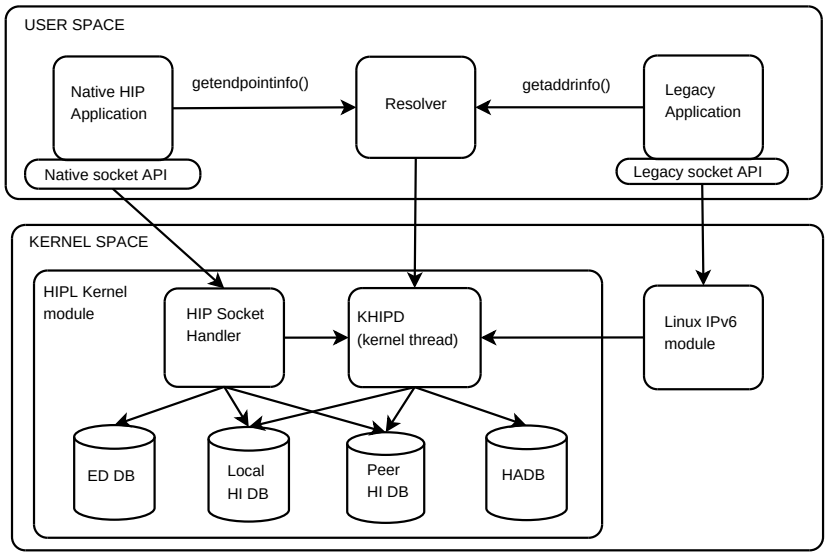

Figure 3.4: HIPL kernel space version architecture

HIP Module

The kernel space component of the HIPL implementation can be either statically compiled into the Linux kernel or it can be loaded into the kernel run-time. The kernel module implements the HIP protocol. For instance, the module handles the base exchange, maintains the HIP associations and takes care of mobility related functionality such as the update messages.

The most important part of the kernel module is the *khipd* kernel thread. The thread sleeps and waits for incoming work orders. All Incoming HIP network packets cause work orders to be created and passed to the kernel thread. The HIP socket handler is included in the kernel module. It handles the HIP specific socket calls that are associated with the HIP native API.

Communication between the user and the kernel space happens via HIP specific sockets. For example, HIP identifiers and mappings of HITs to Ipv6 addresses are communicated to the kernel module by using the socket interface.

Messaging Interface between the Kernel and User space

The user space communicates with the HIP kernel module through the socket interface, specifically by using the PF_HIP family sockets. The information

Output hooks	
hip_get_addr()	Returns an IPv6 address that corresponds to the given HIT.
hip_get_saddr()	Called if we have a packet with a HIT as the destination address and want to find the corresponding src HIT.
hip_handle_output()	Handles outgoing IPv6 packets that have a HIT as the destination address.
hip_get_default_spi_out()	Finds out the SPI to use in an outbound ESP packet.
Input hooks	
hip_handle_esp()	When an ESP packet is received and its SPI belongs to a HIP connection, the IPv6 addresses are replaced with the corresponding HITs before the packet is delivered to ESP handling routines.
Other hooks	
hip_handle_ipv6_dad_completed() hip_handle_inet6_addr_del()	

Table 3.2: HIPL hooks inside the IPv6 networking stack

that is communicated between the kernel and user space contains mostly same kind of data that is enclosed in the actual HIP packets that are sent over wire. Consequently, the messaging interface exploits the same packet format that is used in the HIP protocol messages. In the user space, the communication is carried out by the resolver library and the hipconf tool. On the kernel side, the HIP socket handler receives the messages. Ordinary user space applications do not communicate directly with the HIP kernel module. The only interface are the legacy and native API routines.

Hooks in the Linux Networking Stack

The HIPL implementation uses so called *hook* functions to bypass the normal network packet control flow in the IPv6 stack and pass the control to the HIPL kernel module whenever necessary. There are hooks both in the input

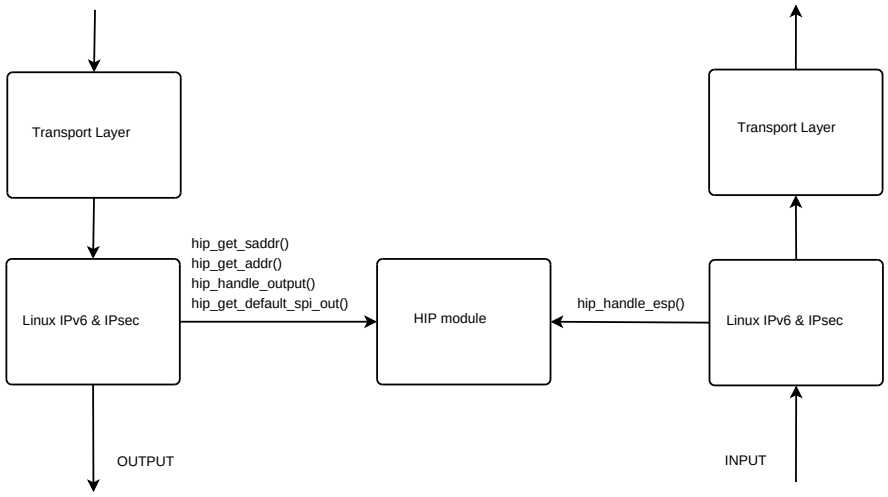

Figure 3.5: HIPL hooks inside the Linux networking stack

and the output side of the IPv6 part of the networking code. The hook routines are itemized in Table 3.2. Figure 3.5 illustrates the hooks inside the Linux networking stack.

Hook functions *hip_get_addr* and *hip_handle_output* are used to convert source and destination HITs to IPv6 addresses in the output side of the IP routing layer. *hip_get_saddr* determines a source HIT for a packet that has a HIT as the destination address. *hip_get_default_spi_out* finds out the SPI value to use in an outbound ESP packet. *hip_handle_esp* replaces the IPv6 addresses with HITs in the incoming packets before they are forwarded to the IPsec module.

Using ESP with HIPL

The HIP module registers two IPsec Security Policies (SP) when it is initialized. A policy is created both for incoming and outgoing HIP packets. Each SP defines a set of IP traffic that is encompassed by the policy [7]. HIP traffic can be matched with the two HIP SPs because it is possible to distinguish between HITs and the plain IPv6 addresses by inspecting the upper-most two bits in each packet [14].

If an outgoing packet with a HIT as the destination address is detected, IPsec is applied because the packet correlates with a HIP SP. Next, the Linux IPsec module tries to find the corresponding IPsec Security Association. If any SA cannot be found for the given destination address, the HIP base exchange is

triggered by the HIP module, specifically by the *hip_handle_output* hook. As the result, two SAs and a HIP association are created.

HIP related ESP transport is implemented by rewriting IPv6 headers below IPsec. IP headers in the IPv6 transport mode packets contain HITs instead of IP addresses. After IPsec processing, all outgoing packets are inspected. If the source or destination address fields contains HITs, they are replaced with the right IPv6 addresses. Incoming packets are handled in a similar way. Before the packets are forwarded to the IPsec processing routines, the IPv6 addresses are replaced with HITs based on the SPI value in the incoming packet. Specifically, the implementation is handled by the *hip_handle_esp* hook on the input side and by the *hip_handle_output* hook concerning the outgoing packets.

Host Association Database

The Host Association Database (HADB) is a global data structure in the HIPL kernel module. It is used for storing information about HIP associations. The HADB contains Host Association (HA) entries. A HA entry includes both parties' HITs, peer's IPv6 address, SPI value for the ESP, HIP state information, shared secret key and some keying parameters.

Specifically, the HADB is a hash table. Peer HITs are used as hash keys. In addition to the HADB, there is a supplementary hash table that is used for mapping SPI values to peer HITs. The second hash table is used when a HA entry has to be found by a SPI value. This is accomplished by first mapping a SPI value to a peer HIT and then mapping the peer HIT to a HA.

Endpoint Descriptor Database

The Endpoint Descriptor Database (ED DB) is used by the HIP socket handler for storing the Endpoint Descriptors. Each database entry contains a *sockaddr_ed* structure and the corresponding HIT. In addition, identification data about the user space process that caused the creation of the ED entry, is included.

Resolver

Resolver is a user space component that handles the HIP endpoint resolution for network applications. Furthermore, the resolver communicates HIs, HITs

and IP address mappings to the HIPL kernel module as a part of the HI resolution.

The */etc/hip/hosts* configuration file contains mappings from peer host names to HITs. Legacy applications convert peer host names to *sockaddr_ in6* structures by calling the modified *getaddrinfo* routine. Native HIP applications use the resolver function *getendpointinfo* that returns *sockaddr_ ed* structures.

HITs are mapped to IPv6 addresses by comparing the canonical hostnames in the */etc/hip/hosts* and */etc/hosts* files. The mappings are communicated to the kernel as a side effect of resolver calls. Thus, the situation where a connection is tried to be established without a preceding resolver call is problematic for the implementation.

The getendpointinfo routine can be used to convert local HIs to EDs. A local ED is returned as a response to a function call where the *nodename* argument is left empty.

HIP Socket Handler

The HIP socket handler is registered into the networking stack when the HIP module is loaded into the kernel. Socket API function calls are passed to the handler if the corresponding sockets are associated with the PF_HIP socket family.

The handler supports Endpoint Descriptors (ED) so that they can be directly used in the socket API function calls whenever the HIP protocol family is set. HIP native API socket calls *bind* and *connect* accept the HIP specific sockaddr_ed structures as input arguments.

The socket handler works as a *wrapper* for IPv6 socket API functions. Its main task is to convert EDs to the related HITs and forward the HIP native API calls to the socket routines in the Linux IPv6 module.

The socket handler processes socket calls that originate from native HIP sockets. When the legacy API is used, the HIP handler is not participating in the data processing. In that case, the 128-bit HITs are handled similarly to the normal IPv6 addresses. HIPL specific *hooks* inside the Linux Ipv6 networking module make sure that the packet processing is forwarded to the HIP module when needed.

The legacy usage of HIPL does not introduce any changes to the IPv6 socket API. Server side applications accept connections to local HIs by binding to sockets with the **in6addr_any** macro. Clients use a HIT in the place of an

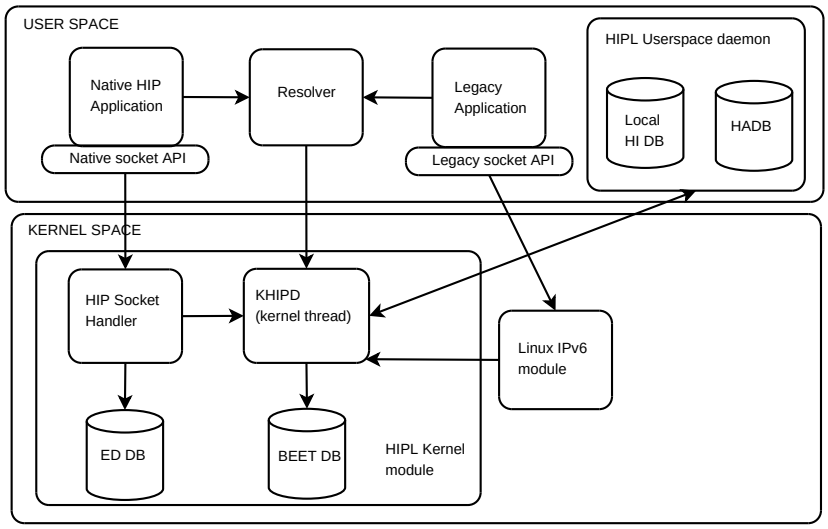

Figure 3.6: HIPL user space version architecture

IPv6 address in order to specify the peer address to connect to.

3.3.2 User space Version

In this section, the main differences between the kernel based and the user space implementations are briefly discussed.

The user space version of HIPL is visualized in the Figure 3.6. In addition to the khipd kernel thread, a user space daemon is now included. Most of the processing related to the HIP protocol is done by the daemon and the kernel module is mainly forwarding messages back and forth.

The user space version introduces some notable changes on the data structures in the kernel. The HADB has been relocated to the user space. Additionally, a host association entry now includes the local HI used for that specific association. An entry contains also function pointers to sign and verify functions which can be used conveniently when handling HIP packets belonging to a specific HIP association. The pointers are set to refer to the correct asymmetric algorithms corresponding to the HI type. Currently, DSA and RSA public keys can be used as HIs.

An auxiliary database has been added into the kernel module. The *BEETDB* is used in the kernel module to enable the HIP - IPsec integration. Actually, the BEETDB replicates some data in the HADB. A BEETDB entry contains

both parties' HITs and a SPI value for one HIP association, for example.

In the kernel based version, all time-consuming cryptographic functions are implemented in the kernel. In the user space version, the OpenSSL library is used to provide the cryptographic routines. All HIP related cryptography is implemented in the user space. Accordingly, also the Host Identifiers are stored in the user space.

More data is flowing back and forth between the user and the kernel space. For example, when a local HI is added, a message is first sent from user space to the kernel module. Then the message is forwarded anew to the HIP daemon in the user space. Thus, there is an unnecessary round-trip. However, this way there is no need to implement a userspace-to-userspace messaging interface for the daemon.

Chapter 4

Requirements and Evaluation Criteria

In this chapter, requirements concerning the implementation are described. Functional and non-functional requirements are discussed separately. Furthermore, evaluation criteria concerning the outcome of the implementation is defined.

4.1 Functional Requirements

To keep the current HIPL implementation interoperable with other implementations it must adhere to the evolving HIP IETF drafts. According to the HIP base specification section 7, a proper HIP implementation must support more than one simultaneous Host Identities [14]. Additionally at least one of the identities should be reserved for anonymous usage. The Draft explicitly recommends support for more than two identities. The Internet Draft also states (in section 3) that all HIP implementations must support the RSA public key algorithm, and should support the DSA algorithm [14].

To put it briefly, it is required that an arbitrary number of Host Identities and the corresponding identifiers should be able to coexist within a HIPL enabled host and function simultaneously. The underlying key algorithm of a HI should not affect the overall functionality. PEM formatted DSA and RSA keys should be usable as HIs.

4.1.1 Key Selection

The legacy version of HIPL was not designed to support multiple Host Identities. The key selection does not work properly in a situation where a host accommodates multiple HIs. For example, R1 messages are precreated only for a single HI at a time. However, the HIP specification states that if the responder holds multiple local HIs, the used responder HIT must match the initiator's request [14]. In other words, when a I1 message is received, the responder should always respond with a precreated R1 that matches the destination HIT in the I1 message.

Briefly, the implementation is required to handle key selection properly. The right local HI should be used when processing HIP messages. Concerning the base exchange, the responder should be able to respond to I1 messages destined to any of the responder's local HIs.

4.1.2 HIP Associations

The legacy HIPL implementation is able to establish and maintain simultaneous HIP associations with more than one host. However, multiple parallel associations between a pair of hosts are not supported.

The HIP specification defines that a host can have more than one association with another host [14]. The implementation should be able to establish and maintain multiple parallel HIP associations between two hosts. Any association can be distinguished by the two involved HITs. Thus, it is not possible to have more than one association between any given HIT pair.

4.1.3 Endpoint Resolution

Endpoint resolution should work both using the native HIP API and also the legacy API. Peer identifiers should be resolved using the **/etc/hip/hosts** configuration file. Only one HIT per peer host is supported in the legacy version of the HIPL implementation. The new requirement is to enable multiple HITs per peer host.

Accordingly, endpoint resolution is required to work for the local identifiers. Applications may call the resolver in order to bind to a specific local identifier. It should be possible to select the right identifier explicitly between the possibly many references returned by the resolver.

4.1.4 Access Control

Legacy applications are not HIP aware. System administrator creates pre-assigned system specific Host Identifiers and stores them into the */etc/hip/* directory on the host. Concerning the legacy use of HIPL, there are no special access control requirements. System HIs should be usable by all processes.

HIP aware applications can use HIP explicitly. The native API allows advanced applications to deal directly with the Host Identifiers. Applications can specify their own identifiers and initiate the HIP base exchange using the specified identifiers.

Regarding the HIP native API, some access control is required concerning the application specified identifiers. Applications should be able to control whether the identifiers are usable by other processes. Access to application specified HIs should also be controlled so that it is not possible to use them erroneously via the legacy API.

4.1.5 User-specific Identifiers

The requirements are related to a multi-user operating system with possibly hundreds of users. The implementation should function as an enabler for user-specific HIP identifiers. In addition, an identity could refer to an user group. In that case, the corresponding identifier would be shared among a group of users. A user-specific identifier could be stored in a smart card, for instance. An application would load the identifier from the card and represent the user in a networked computing node. The actual practical arrangements, such as the establishment of the user-level identities, are out of scope.

4.2 Non-Functional Requirements

4.2.1 Compatibility and Usability

The implementation should not break the existing HIP API design. The modified HIPL implementation is required to be usable both through the legacy API and the native API.

4.2.2 Quality

The Linux kernel isolates user space programs so that each process has its own private address space. Thus, an error in a program cannot normally cause any harm to other running user processes. Inside the kernel space, the situation is very different because the kernel lacks memory protection [12]. In addition, synchronization and concurrency are major concerns within the kernel. Locking must be handled correctly. For the aforementioned reasons, programming errors are almost always fatal inside the kernel space and quality assurance concerning coding is needed.

The HIPL code should not produce memory leaks or null pointer errors. Locking mechanisms should be carefully designed and tested. The code should compile without errors and all warning messages from the compiler must be taken seriously. The HIP base exchange is used as the basic testing tool.

4.3 Evaluation Criteria

Based on the described requirements, two functional conditions that can be used for the evaluation of the implementation, are defined.

1. HIP base exhange should work between two hosts by using any of the multiple local Host Identifiers on either side.

2. n different processes should be able to initiate the base exchange with a remote host by using differing Host Identifiers. It is assumed that each process defines its own identifier.

A successful base exchange is defined as follows. Both participating endpoints start without any state regarding each other. One of the endpoints acts as the initiator and starts the exchange consisting of four messages in total. After having successfully exchanged all four packets, the HIP association must be in the ESTABLISHED [14] state in the both ends.

Chapter 5

Implementation

In this chapter the implementation concerning the multiple identities is described. The architecture of the existing HIPL implementation was not affected radically. The work consists of a number of smaller changes and improvements which provide the new required functionalities for the multiple HIs.

The focus is on the kernel based HIPL implementation. Most of the modifications were also ported to the user space version but since the kernel based approach was the original target, emphasis was placed on it.

5.1 Data Structures

In the following, the most relevant modifications concerning the HIPL data structures are described. The discussion includes rationale for each of the changes.

5.1.1 Host Association Database

The HADB inside the HIPL module was not originally designed to support multiple HIP associations between two hosts. Multiple associations with a peer host could not be established in a situation where only the local HI was varied but the corresponding peer identifier remained the same. This erroneous behavior resulted from the database indexing scheme. The HADB is implemented as a hash table and previously it was indexed by the peer HIT. However, this does not suit the requirements since it should be possible

to maintain multiple simultaneous host association entries related to a single peer host.

An obvious solution was to index the HADB with hash keys created from identifier pairs. The hash value is calculated by applying a hash function on a HIT pair. Each pair of HITs is composed of the local and peer HITs belonging to an association.

Figure 5.1 illustrates two hosts sharing multiple HIP associations. The arrow shape depicts the state of the related HIP association. The corresponding HADB entries are included in the picture.

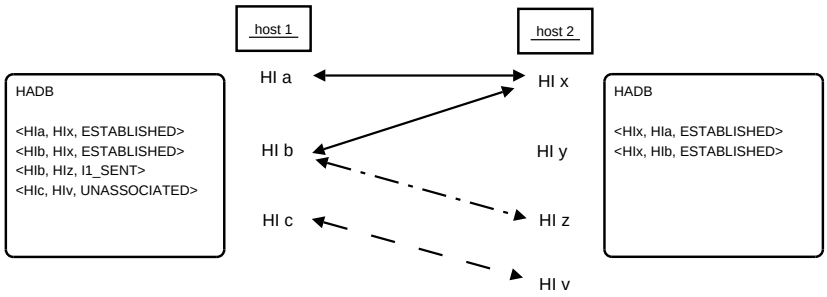

Figure 5.1: Multiple simultaneous associations

5.1.2 Endpoint Descriptor Database

Endpoint Descriptors are handled by the HIP socket handler and they are stored in a database inside the HIP kernel module. Each database entry contains ownership information related to an ED. The corresponding data structure is the *hip_ eid_ owner_ info*. The process ID value, *pid*, was included in addition to the former information about the process UID and GID. Furthermore, an integer data field, *flags*, was added in the data structure. The flags field contains a bit mask that indicates whether an ED binding can be reused by other processes or not.

```
struct hip_eid_owner_info {
        uid_t                   uid;
        gid_t                   gid;
        pid_t                   pid;
        se_hip_flags_t        flags;
};
```

5.1.3 Socket

The *socket* structure is a general data structure for the socket layer in the Linux TCP/IP implementation [5]. The socket structure holds control and state information and it supports BSD type socket interface.

The socket structure was modified so that references to the local and peer EDs were included as two auxiliary integer fields. Specifically, each of the fields contains an integer value that matches the *ed_val* value of a specific ED.

The reasoning behind this modification is related to the HIP native API and to the HIP socket handler. An ED entry is deleted from the kernel database when there are no sockets bound to the ED. Specifically the removal happens when the *hip_socket_release* routine is called. A reference to the associated ED is needed when processing the function call.

```
struct socket {
        socket_state            state;
        unsigned long           flags;
        struct proto_ops        *ops;
        struct fasync_struct    *fasync_list;
        struct file             *file;
        struct sock             *sk;
        wait_queue_head_t       wait;
        short                   type;
        unsigned char           passcred;
        /* BEGIN HIPL MODIFICATIONS */
        uint32_t                local_ed;
        uint32_t                peer_ed;
        /* END HIPL MODIFICATIONS */
};
```

5.2 R1 pre-creation

In HIPL, the R1 messages are pre-created when local identifiers are loaded into the HI database inside the HIP module. More precisely, the messages are generated either when the resolver is queried for the local identifiers or when the hipconf tool is used to add the system identifiers explicitly into the kernel.

The pre-creation of R1 messages was not managed properly for multiple identifiers in the legacy implementation. Pre-created messages were stored in the HIPL kernel module for only one identifier at a time. The possibility to accommodate multiple local identifiers was not taken into account. However, the HIP base specification [14] explicitly states that if the responder has multiple HIs, the packet's source HIT is required to match the initiator's request.

The problem was approached by deploying a new hash table for the pre-created R1 messages. This solution enables pre-created R1s for multiple local HIs since HITs can be used as hash keys. During the base exchange, the responder can read the destination HIT from the received I1 message and use it to find the right entry from the R1 hash table.

In the user space version the pre-created R1 messages are included in the HI database together with the corresponding local identifiers. Thus, there is no need for an auxiliary hash table.

5.3 Bind

Formerly it was not possible to offer a HIT to the socket API *bind* function because IPv6 code would normally return an error when doing address type checks against HITs and reject them. This held both for the legacy and the native HIP sockets. The *in6addr_any* macro [4] was utilized to bypass the problem. The macro denotes that operating system is let to select the socket's source address. In consequence, the legacy implementation did not allow to associate sockets with specific Host Identifiers. One could not choose the HI to use in the initiating I1 message either.

The described legacy solution does not work for multiple identifiers scenario, since it is required that applications are able to bind sockets to specific local identifiers. Both a server and a client socket can be associated with a HI.

Consequently, modifications had to be done to the IPv6 part of the Linux networking stack. HITs had to be handled separately from the normal IPv6 addresses. The code was modified so that the IPv6 related address verification is bypassed in the *inet6_bind* routine if a HIT is encountered instead of a normal IPv6 address. This way, also HITs can be passed to the bind function.

A certain kind of address verification is needed concerning HITs, as well. When a HIT is tried to be bound to a socket, one must make certain that the

corresponding Host Identifier does really exist locally. In addition, a check is needed to ensure that the currently running user space process has the right to access the specified HI. The existence check and the access right verification are handled by a new hook routine called *hip_check_access_to_local_hit*.

5.3.1 Using the Bind Routine

Due to the modifications described in the previous section, processes are able to bind sockets directly to HITs.

Figure 5.2 visualizes the binding sequence regarding a native server application and a connection-based socket. The sequence starts when the server binds an ED address to a previously created socket. The HIP socket handler receives the bind call and converts the ED to the corresponding HIT. Next, the handler forwards the bind call with the resolved HIT to the Linux IPv6 bind routine. Access rights to the local HI are checked with the *hip_check_access_to_local_hit* hook function. The server application proceeds by starting to listen for incoming connection attempts.

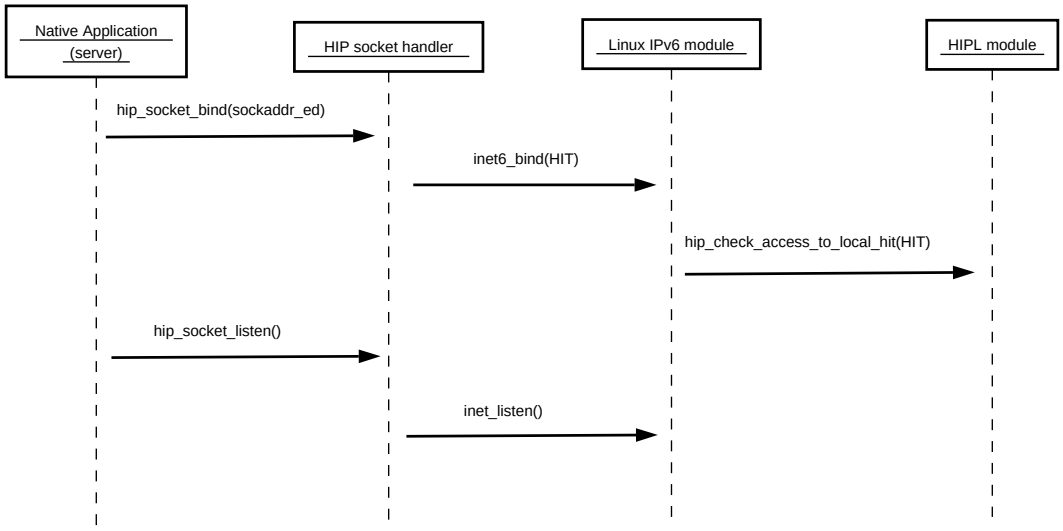

Figure 5.2: Binding a local HI to a server socket

Also client applications may want to bind a socket to a local identifier. Figure 5.3 depicts the situation from the point of view of the native socket API. A client can initiate the HIP base exchange with a specific local HI by binding the corresponding HIT to a socket and calling the connect routine after that.

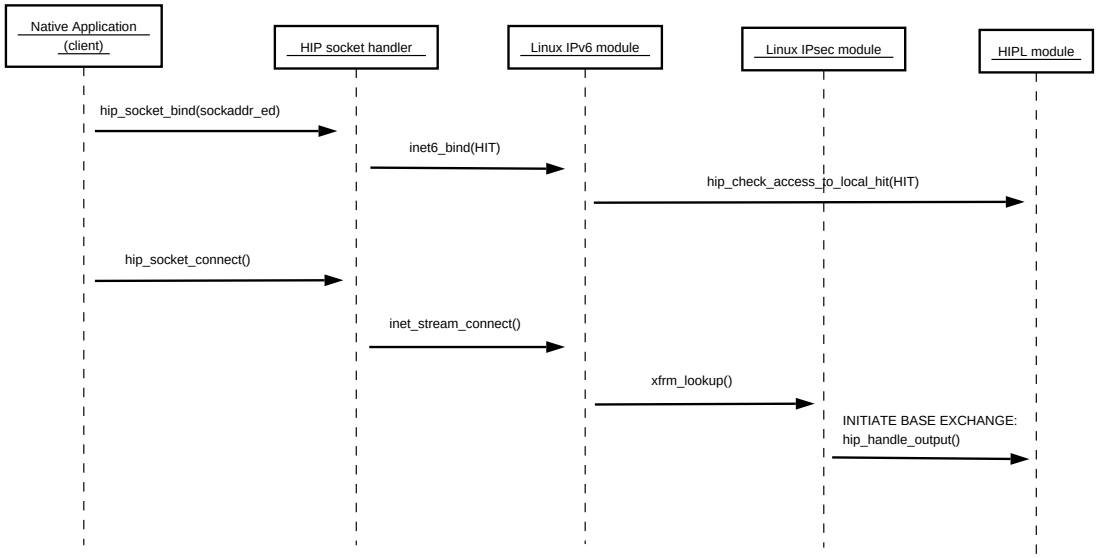

Figure 5.3: Binding a local HI to a client socket

The binding sequences are visualized only regarding the native HIP sockets. However, the function calls regarding the legacy socket API are almost analogous. The only significant difference is that the HIP socket handler is bypassed altogether when using the plain IPv6 sockets.

5.4 Hooks

Section 3.3.1 described the *hook* functions that are used to incorporate the HIP processing to the Linux IPv6 module. The hooks allow to bypass the normal network packet control flow in the IPv6 stack and pass the control to the HIPL kernel module whenever HITs are detected in the place of IPv6 addresses. Some changes were required in the hook routine implementations due to the new indexing scheme in the HADB.

The *hip_get_saddr* hook is called during the IPv6 output processing in order to find out a source HIT that corresponds to a specific destination HIT. When a client application explicitly binds a socket to a local identifier the hook becomes actually unnecessary due to the enhancements done in the bind routine. In that case, both the source and destination HITs are already known when the hook routine is processed. The situation is different when a client does not bind to any local HI explicitly but still tries to connect to a

remote destination HIT. In that case, the used local HI is selected at random. The first association with a matching destination HIT is looked up from the HADB and the related source HI is utilized in HIP packets. The usage of some local identifiers can be restricted. In consequence, access permissions are checked during the procedure. A more detailed discussion about the HI related access control is included in 5.6.3.

In addition to the changes in *hip_ get_ saddr*, minor modifications were needed concerning the hooks *hip_ handle_ output*, *hip_ get_ addr* and *hip_ get_ default _ spi_ out*. The routines make use of the HIP association data that is stored in the HADB. Changes were needed in order to adapt the hooks to the new indexing scheme. HADB entries are accessed by hash values calculated from HIT pairs. Formerly, the associations were looked up by using only the peer HIT in the query.

5.5 Resolver

Applications commonly connect to peer hosts by using network host names instead of using IP addresses, or in this case Host Identifiers, directly. The host names are converted to HIP layer identifiers by calling functions in the *libinet6* resolver library.

Changes were needed in the resolver implementation and also in the kernel module to handle the multiple peer and local identifiers correctly. Both the legacy applications and HIP aware applications had to be taken into account.

The *getaddrinfo* resolver function had to be modified for the legacy API usage. HIP aware applications on the other hand required modifications in the *getendpointinfo* routine. The endpoint resolution was enabled both for peer and local identifiers.

In the new scenario multiple peer HITs are enabled in the */etc/hip/hosts* configuration file which is used to resolve the Host Identity Tags of known HIP enabled peer hosts. To be more specific, a line in the file includes a host name and one or many HITs that are associated with the name. This way an application may receive multiple *sockaddr_ in6* or *sockaddr_ ed* structures from the resolver if a specific host name is associated with multiple HITs.

In like manner, endpoint resolution was enabled for multiple local identifiers. Both the legacy API resolver routine getaddrinfo and the native API function getendpointinfo were modified to support local identifiers. Both functions resolve local HIs from the */etc/hip* directory when the *node* argu-

Function	Identifiers	*node* arg	*service* arg	Resolved addresses
getaddrinfo	local HIs	NULL	port number	*sockaddr_in6*
getaddrinfo	peer HITs	peer name	NULL/port	*sockaddr_in6*
getendpointinfo	local HIs	NULL	port number	*sockaddr_ed*
getendpointinfo	peer HITs	peer name	NULL/port	*sockaddr_ed*

Table 5.1: Summary of the resolver routines

ment is left NULL. The system-specific identifiers are distinguished by the file names which must follow the wildcarded string pattern *hip_host_*_key**. For example, DSA private keys might be named as *hip_host_dsa_key1*, *hip_host_dsa_key2* etc.

Resolver functions are summarized in Table 5.1. Legacy applications use the getaddrinfo function that returns *addrinfo* structures containing ordinary IPv6 socket addresses. Native applications call getendpointinfo and receive one or many *sockaddr_ed* embedded in *endpointinfo* structures.

5.5.1 Mappings

The resolver receives a peer host name and maps the name to both a HIT and an IPv6 address. The HIT and the IPv6 address are communicated to the kernel module as a part of the resolver call. The resolver functions work similarly in this sense both for the legacy and the native API.

Formerly, only one HIT to IPv6 address mapping was added into the kernel's Host Association Database per a resolved peer HIT. A resolver call created a fresh HADB entry containing the peer host's HIT and IPv6 address. However, a network node can now host a multitude of Host Identifiers unlike before. In addition, the indexing scheme of the HADB was changed as a part of the implementation. A single HADB entry per a peer host is not sufficient when multiple associations can be formed between two hosts. For these reasons, the mapping process had to be refactored.

The HIPL kernel module needs to know the network address of the peer when the socket API connect routine is called. The I1 message of the HIP base exchange must be sent to the right network address. The HADB is used for finding the peer address that matches both the initiator's and the responder's HIT. Since the resolver cannot know beforehand the specific local HI that is used to initiate the base exchange, an association per every local HI is needed. Specifically, one HIT to IPv6 address mapping creates a new

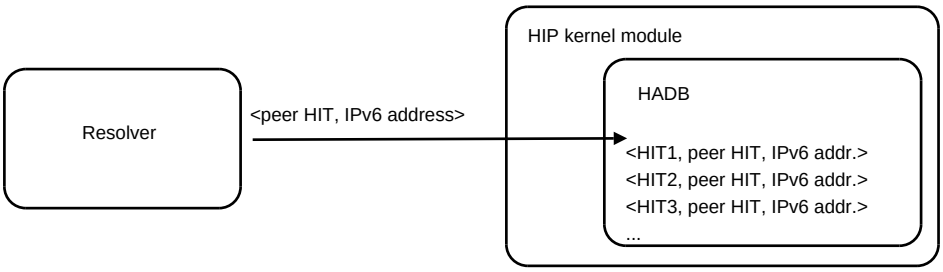

Figure 5.4: HIT-to-IPv6-address mapping with multiple local HIs

HADB entry per a local identifier in the HI database. Figure 5.4 visualizes the mapping procedure.

5.6 Endpoint Descriptors

Endpoint Descriptors are allocated by the HIP socket handler. User space processes can access EDs by calling the HIP native API functions *getendpointinfo*, *getlocaled* and *getpeered*. Native HIP sockets represent the PF_HIP family and accept socket addresses in the form of *sockaddr_ed* structures. Every sockaddr_ed embeds an ED number, *ed_val*, that is set inside the socket handler and used to reference the entries in the kernel's ED database.

5.6.1 Local Endpoints

Local Endpoint Descriptors serve as references to the identifiers that identify the local host. The Native HIP API Draft defines three macros that enable the reuse of the local endpoints [9]. HIP_HI_REUSE_UID and HIP_HI_REUSE_GID allow the HI binding to be reused by processes with the same User ID (UID) or Group ID (GID) than the calling process. The third, HIP_HI_REUSE_ANY, allows any process to use the ED. The reuse flags were originally designed for the *getlocaled* routine that is used to convert application specified endpoint structures to ED socket address structures [8]. The reuse functionality was implemented as a part of the multiple identities implementation.

The resolver functions *getendpointinfo* and *getaddrinfo* resolve local identifiers from the */etc/hip* system directory when the *node* argument is left NULL. Both functions communicate the endpoint structures to the HIP

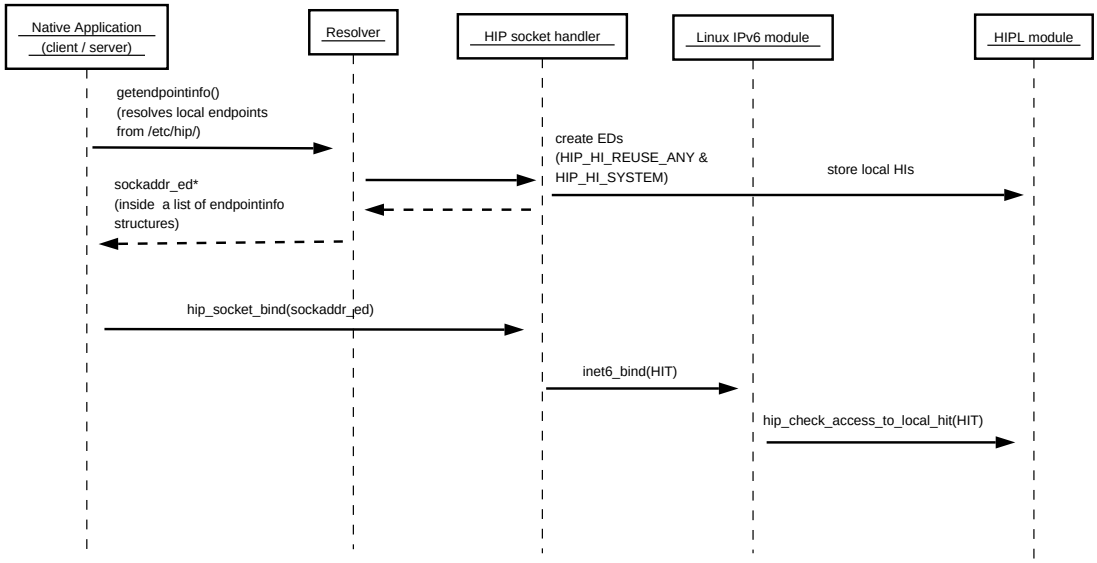

Figure 5.5: Binding to a system HI using the native HIP API

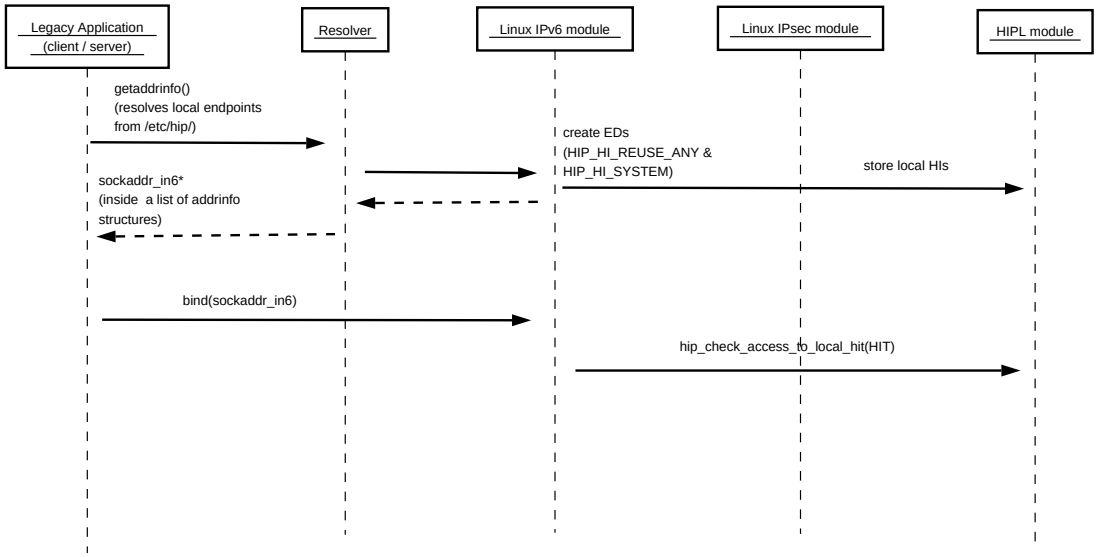

Figure 5.6: Binding to a system HI using the legacy HIP API

socket handler through the socket interface. The handler receives endpoints structures from the user space resolver routines and creates new EDs that correspond to the local HIs. Both the getaddrinfo and getendpointinfo calls are handled similarly inside the socket handler. However, only the native

API function getendpointinfo returns EDs to the calling userspace process.

In order to ensure that the system identifiers are usable by all processes, they are handled differently from other identifiers. All system HIs are automatically converted into EDs that are equipped with the HIP_HI_REUSE_ANY flag. In addition, the flag HIP_HI_SYSTEM is used to indicate that the associated ED entry is for a preassigned system-wide HI. The conversion from a system HI to an ED is handled this way both in the getendpointinfo and in getaddrinfo. In addition, the same scheme holds for the *getlocaled* that is used to convert application specified HIs into EDs. To ensure that a process cannot reserve system HIs exclusively for itself, all identifiers that are loaded from the /etc/hip/ directory with getlocaled are automatically associated with both the HIP_HI_REUSE_ANY and the HIP_HI_SYSTEM flag.

In Figure 5.5 a native application resolves the local system identifiers and binds a socket to one of them. Figure 5.6 visualizes the same sequence for a legacy application.

5.6.2 Endpoint Descriptor Life Cycle

Although a single HI can conceptually relate to a multitude of EDs at a time, the implementation restricts this relationship to one-to-one. This limitation was chosen purely for practical reasons. Currently, there is no need for more complicated design. A single local ED can be used by multiple processes if the owner process has enabled one of the REUSE flags. Peer EDs are always usable by all processes. Furthermore, the one-to-one relationship makes possible to use EDs as the basis for HI access control.

The HIP socket handler inspects the ED database whenever either *getendpointinfo*, *getlocaled* or *getpeered* is called. Each of the functions communicates a number of HIs or HITs to the handler through the socket interface and return EDs to the caller. The socket handler conducts a database search based on HITs that are received from the user space resolver functions. A new ED entry is created if the handler cannot find an existing entry with a matching HIT.

Actually, EDs are processed inside the kernel also when the legacy API routine *getaddrinfo* is called to resolve the local addresses. The HIP socket handler receives local HIs from the user space and creates the related EDs if they do not exist already. Although the legacy API does not allow to handle the EDs directly, local HI resolution is processed internally in a similar manner to the other resolver functions. This approach was chosen to simplify the implementation.

A new ED entry and a unique *ed_val* value are generated when the HIP socket handler receives either a local identifier or a HIT corresponding to a peer host. In addition, the user, group and process ID values of the calling user space process are stored in the newly allocated database entry. Further, any flags that were associated with the HIP API function call, are copied to the entry.

Each ED entry includes a reference counter that allows tracking the number of processes that are using the same entry at the same time. The reference counter is initially set to 1 and it is incremented every time a new process reuses an ED by calling either *getlocaled* or *getpeered*. Accordingly, the reference counter is decremented when a process stops using an ED. Specifically this happens when the corresponding native socket is released or when the socket handler fails to bind an ED to a socket because of a failed access right check.

A ED is deleted when the reference counter drops below 1. In the case that the ED is used simultaneously by multiple processes, the deletion happens when the last binding socket is released. The EDs that correspond to the local system identifiers are handled differently. Their existence is not dependent on the socket bindings because the system HIs are likely to be used very frequently by multiple processes. Consequently, the related EDs are removed only if the user space tool *hipconf* is utilized for explicit deletion.

The life time of an application specified identifier in the kernel's HI database is related to the existence of the corresponding ED. An identifier is removed from the database when the corresponding ED entry ceases to exist. Figure 5.7 depicts the life cycle of an ED and the related HI entry in the kernel. A native HIP application specifies an identifier, binds a socket to the HI by using the related ED, transfers some data over HIP and finally closes the socket. The ED is created during the processing of the getlocaled routine and it is deleted when the associated socket is closed.

5.6.3 ED Based Access Control

In UNIX based operating systems every process has unique process identification value (PID). In addition, information about the related user ID (UID) and group ID (GID) is associated with every running process. The implementation of access control related to the local Host Identifiers is based on the UID, GID and PID values. Further, the reuse flags (see 5.6.1) are used to determine whether a process is allowed to bind to a certain HI.

HI access control is related especially to the local identifiers that are allo-

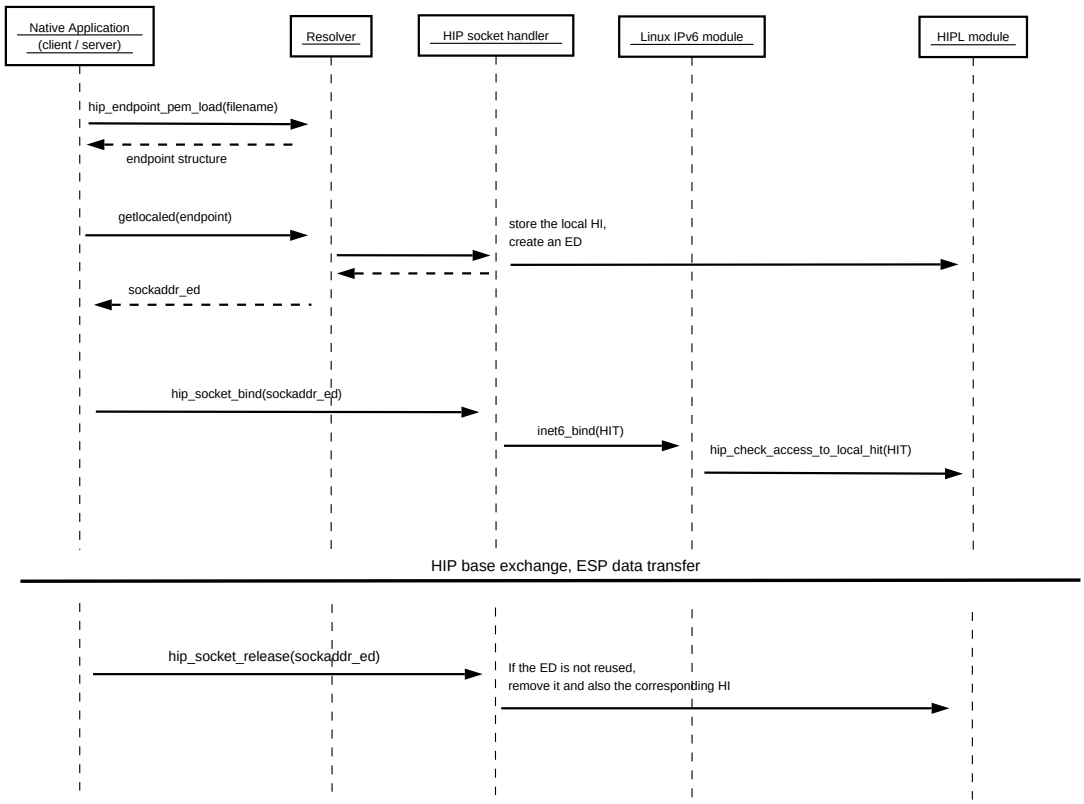

Figure 5.7: Life cycle of an application specified identifier and the correlated ED

cated by advanced HIP aware applications through the native API functions *hip_endpoint_pem_load* and *getlocaled*. The implementation allows to restrict binding to the application specified identifiers that are already loaded into the kernel. The resolver routines *getaddrinfo* and *getendpointinfo* are related to the system-wide HIs which are usable by all processes by default.

Access control decisions are based on the information in the ED database. Every database entry includes *eid_owner_info* structure which contains information about UID, GUID and PID of the process that allocated the entry. In addition, the structure includes a data field for storing possible reuse flag values. When a process attempts to bind a local ED to a socket, the ownership information is checked from the ED database. The PID value of the binding user space process is compared to the value stored in the related ED entry. The PIDs must match if any reuse flag is not set. If the reuse of an identifier is enabled by using one of the special macros, the GID and UID

Flag	Binding allowed to
HIP_HI_REUSE_ANY	All processes
HIP_HI_REUSE_UID	Specific user (UID)
HIP_HI_REUSE_GID	Specific user group (GID)
-	Specific process (PID)

Table 5.2: Access control matrix

values are first inspected for making the access right decision. Access to the local identifiers is denied from unauthorized processes. This simple access control model is visualized in Table 5.2. The related decision-making process is depicted in Figure 5.8.

Legacy application may attempt binding to local identifiers by using the corresponding HITs in the place of the IPv6 addresses. In this case, the binding to a HI is allowed only if the associated ED entry happens to be associated with the HIP_HI_REUSE_ANY flag.

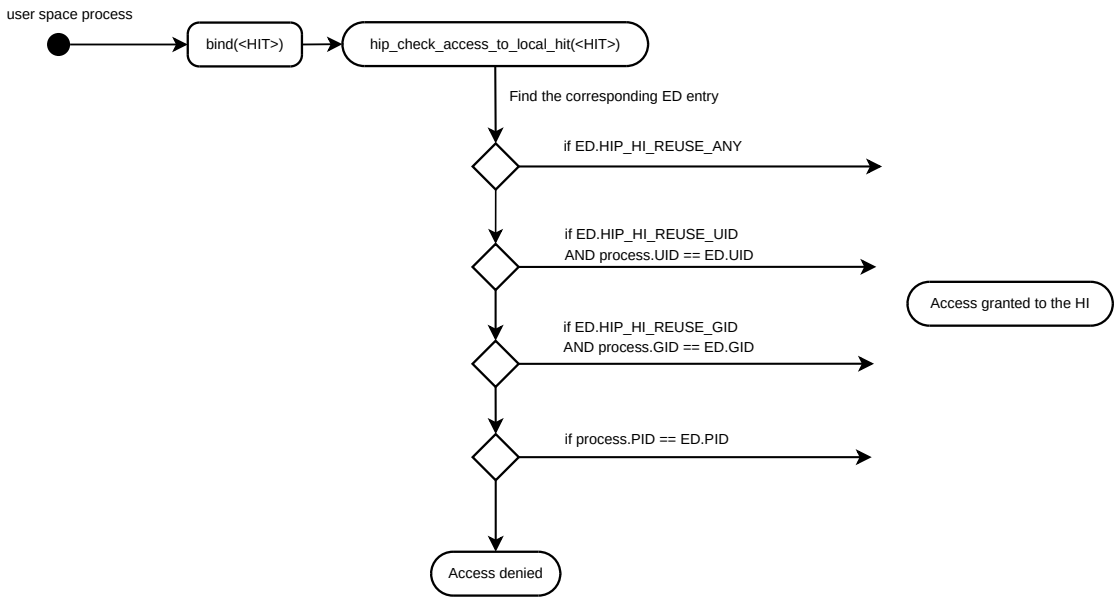

Figure 5.8: Acces control sequence

Access control to local HIs is implemented by checking access permissions within the socket API function *bind*. A new hook routine called *hip_check_access_to_local_hit* was placed in the *inet6_bind* function in the Linux IPv6 networking module. The hook takes care of the access right verification

for both the legacy and the native API bind calls.

5.7 Hipconf

Hipconf is a user space program that can be used to create new identifiers and load them into the kernel's HI database. In addition, HIT to IPv6 address mappings can be communicated to the kernel using the tool.

Hipconf was enhanced so that local identifiers can be explicitly deleted from the kernel. The socket interface is used to communicate the HIT to be deleted to the HIP socket handler which handles the actual removal of the corresponding HI. When a local HI is deleted, also the related Endpoint Descriptor, HIP associations and pre-created R1 messages are removed.

5.8 Public and Anonymous Identifiers

The implementation enables public and anonymous identifiers by holding the related data for every local identifier. It should be noted, however, that this implementation detail might be interesting mostly from the future point of view since the global infrastructure for publishing Host Identifiers is still missing.

Every HI entry in the kernel database includes a bit flag that indicates whether the identifier is anonymous or not. All identifiers that are loaded from the file system can be specified to be either anonymous or public. If the character string *pub* is included in a key file name, the corresponding HI is defined to be a public one. Otherwise, any given identifier is considered as anonymous one.

Chapter 6

Analysis

In this chapter, the outcome of the implementation, described in Chapter 5, is evaluated against the requirements. In addition, potential implications of supporting multiple simultaneous HIs are discussed. Further, some ideas concerning the resolver and the related address mapping are identified.

6.1 Functional Analysis

It is somewhat problematic to evaluate non-functional properties of the solution because it is basically an enhancement to the existing legacy implementation. In addition, the implementation does not form any clearly distinct piece of software but it is instead tightly integrated into the existing code base. In consequence, evaluation concerning the non-functional characteristics, such as performance or robustness, is omitted and the main focus is set on the functional testing and analysis.

6.1.1 Test Environment

VMWare Workstation 4.5.2 was used to test the implementation. The software allows to run multiple x86-based operating systems simultaneously on a single PC. In this case, two virtual hosts running Debian Linux with 2.6.11 kernel were used. IPv6 was enabled for network communication between the hosts. In each of the tests, one host acts as the HIP initiator and the other one plays the responder role.

The test results were verified by inspecting the kernel log messages included in */var/log/kern.log*. In addition, the Linux *proc* file system was used to

have a view of the system state. In Linux, the */proc* directory contains a hierarchy of special files which represent the current state of the kernel. Regarding the HIPL implementation, the */proc/net/hip/sdb_state* file shows information about all HIP associations that are stored inside the kernel. The */proc/net/hip/lhi* file lists all HIs that are loaded into the kernel database.

6.1.2 Multiple Identifiers

The legacy implementation relied on the fact that only one HI could exist locally at a time. The new requirement was to enable multiple identifiers on the local host.

A hash table was introduced for storing the pre-created R1 messages for any given local HI. In addition, the key selection in the base exchange was enhanced so that any of the local HIs can act as the receiver endpoint for the base exchange. The responder reacts to an incoming I1 packet by sending a R1 message that contains the local HI which was used as the destination address in the received I1.

A test application is included in the Appendix A.1. It is utilized together with the test server, listed in Appendix A.3. The test client first resolves all the server host's HITs by using the legacy API *getaddrinfo* resolver routine. Next, the client initiates the HIP base exchange with each of the resolved HIP endpoints in a row. The server host reacts to each initiating I1 message by returning a differing R1. As a result, a distinct HIP association is established with each of the server's HIs.

6.1.3 Multiple HIP Associations

The possibility to accommodate multiple identifiers on a host may result in multiple HIP associations between a pair of hosts. In order to establish and maintain the associations properly, the indexing of the HADB hash table was refactored so that HIT pairs are used as hash keys. Any given HIP association can be distinguished by its respective HITs.

The functionality of multiple associations was verified with the test code described in Appendix A.2. The test setting is very similar to the one that was described earlier related to the multiple identifiers. The most significant difference is that now the client first resolves all local HIP endpoints by calling the native API *getendpointinfo* routine. Next the client connects to the remote test server by using each of the local HIs in a row. The base

exchange is thus initiated multiple times and a HIP association is established per a client identifier.

6.1.4 Resolver

The requirements included endpoint resolution for multiple local and peer identifiers. The user space resolver library was enhanced accordingly. Multiple local and peer endpoints can be queried by using either the legacy or the native API.

Appendix A.4 includes test programs for both the *getendpointinfo* and *getaddrinfo* resolver routines. The functions are tested by querying both the local and peer endpoints in turn. A query for the local endpoints successfully returns an address structure per a HI that is located in the */etc/hip* directory. When the peer endpoints are resolved, the functions correctly return as many socket address structures as there are HITs associated with the peer host name in the */etc/hip/hosts* configuration file.

6.1.5 Controlling Access to Application Specified Identifiers

By default, only the process that loaded an application specified identifier into the kernel is allowed to bind to it. In addition, the HI is removed from the kernel when the binding process releases the associated socket. The native API routine *getlocaled* is used to load an identifier into the kernel. If the function call is associated with one of the *HIP_HI_REUSE** flags, the HI binding can be reused.

A test program for verifying the access control related to HI bindings is included in Appendix A.5. The test consists of a simple server program that loads a HI from a specified key file, binds a server socket to the HI and starts to wait for incoming connections. The program uses the flag parameter (See Table 5.2) to specify whether other processes can bind to the HI.

Testing was conducted by first starting the test server using a specified key file and letting the program to run. Next, another instance of the same program was started with the same key file but with a different port number. The second process is either denied or allowed to bind to the specified HI. The correct functionality of the *HIP_HI_REUSE** flags was verified by varying the used flag value and the UID and GID values of the processes. The latter was accomplished by varying the Linux user entity on whose behalf a process

is running.

6.1.6 Evaluation Against the Criteria

Criteria for functional evaluation of the implementation were presented in Chapter 4. Two distinct functional conditions were described. The first one required that the HIP base exchange should succeed by using any combination of HIs between two communicating hosts. The second condition stated that an undetermined number of distinct processes should be able to initiate the base exchange by using a differing HI. On the grounds of the conducted tests, the implementation meets the requirements.

6.2 Consequences of Hosting Multiple HIs

The support for a multitude of local HIs raises some interesting questions from the research point of view. What kind of opportunities do the multiple coexisting Host Identities open? What are the differences compared to the situation where there would be only one identifier assigned for a computing node?

The possibility to accommodate multiple identifiers on a single host allows to logically separate the entities sharing the host by assigning them with distinct HIs. The concept of *application specified identifier* and the related access control scheme guarantee that the assigned HIs can be reserved solely for the selected entities. An entity might refer to an application, a virtual host or a user, for example. Potentially, also a physical networking interface could be associated with a specific HI but this would require further work.

A HIP identifier can be linked to a specific user by allowing access only for the processes with a certain UID. Users could be automatically linked to distinct HIP layer identifiers by establishing user-specific HIs in the user home directories. However, it is not clear whether this should be actually done. It is questionable whether HIP should resemble the SSH protocol which lies in the application layer and exposes identifiers directly to the users by establishing per-user keys. HIP works under the transport layer and therefore it might not be reasonable to link HIP identities directly to the user-level identities. Instead, HIP layer could be rather invisible to the actual users and another layer of identifiers could be introduced for the user-specific identities, for example.

Further, the HI utilization policy might have a significant effect on the CPU

usage. Even a single key exchange transaction is relatively costly in time and computing resources. If a distinct HIP identifier was automatically established per user within a multi-user OS, a considerably larger number of the HIP base exchange procedures could take place compared to the situation where only a relatively small number of common HIs were used in the network communication. In the latter scheme, the HIP associations would be shared and reused by a number of users and the related processes. In the former scenario, a new HIP association would be established every time a new user communicates with a remote peer host even if some other user had already established an association with the same peer.

On the other hand, it could turn out useful to associate individual users directly with user-specific keys under some specific circumstances. For example, user-specific rules could be configured on a HIP enabled firewall [8]. This could make sense for a company with a band of travelling employees and a need to communicate with them in a secure manner.

In addition, user-specific identifiers could allow to survey the network communications conducted by an individual user to a certain extent. Providing the observer had information about the corresponding HIs, it could be possible to know not only which computer committed a network transaction, but also which person handled the transaction. In this sense, the HI usage policy would turn out to be an *accountability* issue.

6.3 Security Implications

6.3.1 Local Point of View

It is very probable that most of the network applications do not need a direct access to the Host Identifiers. In common cases, it is sufficient that the operating system uses a couple of system-specific identifiers by default for all HIP connections. Identifiers are loaded into the kernel by the resolver and used in the HIP processing inside the kernel invisibly to the application layer. Applications deal with *sockaddr* structures with IPv6 addresses replaced by HITs and never come into contact with the actual HIs.

Advanced applications may want to manipulate identifiers directly. Besides the pre-assigned system identifiers, additional identifiers can be specified and loaded into the kernel with the HIP native API. Processes are able to create their own cryptographic keys and use them as HIP layer identifiers.

Standard Linux file access scheme allows to control which users can access

identifier key files. The file access permissions of any given key file dictate whether the identifier can be retrieved by the *hip_endpoint_load_pem* routine and thus used as an application specified HI.

The implementation allows to restrict access to the application specified identifiers that are already loaded into the kernel. A process may attempt to bind a socket to a HI that was previously loaded into the kernel by another process. Access control is carried out by comparing the PID value of the binding user space process to the value stored in the related ED entry. If a reuse of an identifier is enabled by using one of the special macros, also the GID and UID values can be used for making the access right decision.

This simple access control model for local Host Identifiers seems to suffice for the moment. Both the general-purpose freely accessible system HIs and the application specified identifiers with restricted access are enabled. However, HIP standardization is still an ongoing work. In addition, HIP lacks specifications for managing the local identifiers securely. Any practical experience about managing the HIP identifiers is not yet available either. Only real-world application usage might eventually show whether the current model is adequate or whether a more fine-grained security for HIs will be needed.

6.3.2 Network Point of View

Ylitalo et al. see that using public keys as identifiers is problematic regarding the privacy of a network endpoint's identity [19]. Concerning HIP, a HI identifies a network endpoint. Now, if a host holds only a single HI and uses it repeatedly, it might be relatively simple to scan network communications and detect all traffic that is carried out by the host.

The situation changes when a host can accommodate multiple Host Identifiers. Some of the HIs can be temporary and they might be used only for a limited number of transactions. At the same time there may be public HIs that are published through HIP enabled DNS and are thus more permanent in nature.

A host might attempt to protect its privacy and keep itself unrecognized by allowing only anonymous HIs to be used. The host's operating system could be configured so that all applications were forced to use only temporary identifiers for maintaining the privacy. However, if a public HI was suddenly used for an unanticipated reason, this could potentially lead to the exposure of the host's identity.

6.4 Peer HIT to IPv6 Address Mappings

A network application addresses a peer host by using the peer's host name. The host name is mapped to both a HIT and an IPv6 address by using a resolver call. The resolved address pair is then communicated to the kernel module by the resolver implementation. In consequence, the kernel module has the information about the proper IPv6 address when a client application tries to *connect* to a distant peer by using its HIT.

The described solution for mapping HITs to IPv6 addresses creates unnecessary host associations that are not really used (see 5.5.1). At the moment the unnecessary associations do not cause any problems but the situation might change if a host would accommodate a multitude of distinct Host Identifiers. In order to optimize the memory consumption inside the kernel space, the unused associations could be removed at certain time intervals. This could be accomplished by a special kernel thread or a kernel timer, for example. An alternative solution would be to refactor the resolver or change the way the HIT-to-IPv6-address mappings are stored inside the kernel so that the unnecessary associations would be avoided. In the following sections, these alternative approaches are briefly discussed.

6.4.1 Alternative Resolver Model

Komu describes an alternative resolver model in his thesis [8]. In the alternative model, the responsibility for mapping the identifiers and the locators is delegated from the resolver to the application. The alternative design is depicted in Figure 6.1.

In the case of multiple identities, the application should communicate HIT pairs, consisting of peer and local HITs, together with the peer's IPv6 address to the kernel module. An explicit way to associate a local identifier with a peer HIT would be needed in order to avoid the creation of unnecessary associations. Consequently, a new native API routine might be necessary for this approach.

The downside of this solution is that it would require changes in the syntax of the HIP native API. In addition, it could be very difficult or almost impossible to accomplish the same functionality via the legacy API without breaking the syntax of the standard socket API.

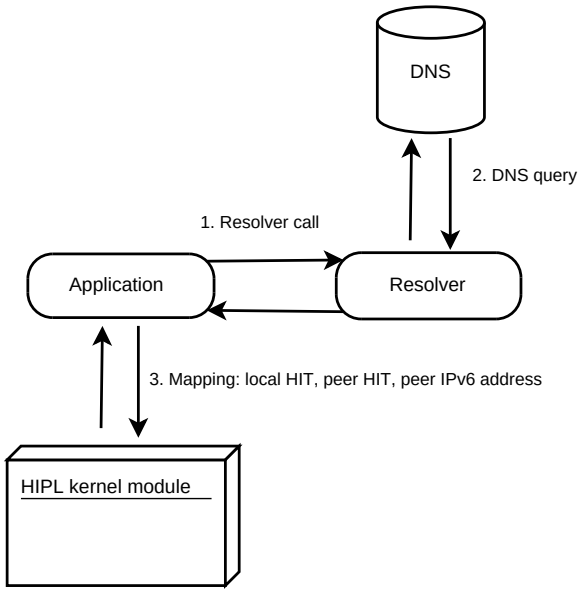

Figure 6.1: Alternative Resolver Model

6.4.2 Separating the Host Associations from Mappings

Formerly, the HIPL implementation handled the initial mapping from a HIT to an IPv6 address by creating a single host association entry as a side effect of the HIP endpoint resolution. The entry did not contain information about the local identifier since only one local identifier was usable at a time. Currently, the situation differs so that host association entries are distinguished by HIT pairs consisting of both the local and peer HITs. Consequently, one host association entry is created per each local identifier during the initial IPv6-address-to-HIT mapping.

One could avoid the creation of the unnecessary associations by postponing the creation of HADB entries until the client end actually calls the *connect* routine and therefore also triggers the HIP base exchange. The initial mappings could be stored into an auxiliary data structure instead of holding them in the HADB. A fresh HADB entry would be created only when the I1 message is actually sent. At this point, the kernel module is aware of the corresponding local identifier and a distinct association entry can be properly created.

Chapter 7

Future Work

The implementation effort brought up some ideas regarding further HIPL development and future research. The most relevant topics are discussed briefly in this chapter.

7.1 HIP Policies

When a host accommodates multiple simultaneous HIs, the HIP key exchange might be iniatiated by using a different local HI for different responders. The HIP specification remarks that all implementations should provide an access control list (ACL) for controlling the local HI usage per destination HIT [14]. Accordingly, an ACL would be needed to dictate whether the receiver side accepts a HIP base exchange with a certain host. Having multiple identities enabled, it could be useful if the HIPL implementation would include such a functionality.

7.2 Missing Native API Functionalities

The HIP native API [9] defines some functionalities that are related to the multiple identifiers but were not included in the implementation. These non-essential features are briefly described in the following.

The API defines functions that might be useful for applications that receive multiple EDs from the resolver. The routines *getlocaledinfo* and *getpeeredinfo* return the HI and the set of IP addresses that are associated with the given ED.

Some unimplemented macros are related to the HIP native sockets. Three macros are important concerning the multiple identities: HIP_HI_ANY, HIP_HI_ANY_PUB and HIP_HI_ANY_ANON. The idea behind these macros is similar to the IPv6 socket API wildcard addresses INADDR_ANY and IN6ADDR_ANY_INIT [4]. The HIP layer macros are useful for server applications that want to accept HIP connections to all available HIP endpoints [8]. With HIP_HI_ANY macro a socket accepts connections destined for any of the local HIs. The HIP_HI_ANY_PUB or HIP_HI_ANY_ANON are associated with a server socket that accepts connections to public or anonymous identities.

Besides the aforementioned socket macros, the native API defines three flags for the *getendpointinfo* resolver routine. If either EI_HI_ANY, EI_HI_ANY _PUB or EI_HI_ANY_ANON flag is passed to the resolver, it returns either an arbitrary local ED or one that is related to a public or anonymous HI.

7.3 Opportunistic Mode

In the opportunistic HIP the initiator starts the base exchange by sending a special I1 message with NULL destination HIT. The current implementation needs modifications in order to enable the opportunistic I1s. The HADB is indexed by HIT pairs consisting of the source and the destination HITs for a HIP association. Some sort of new scheme has to be designed to maintain a HIP state with an unknown destination HIT.

In order to enable the opportunistic mode in HIPL, modifications are needed for the user space resolver, because the socket API *connect* function cannot be utilized for an unknown destination HIT. Regarding the opportunistic HIP, the resolver could use some sort of temporary HIT value when committing the initial HIT-to-IPv6 mapping for an unknown destination HIT. The temporary HITs should contain the normal HIT prefix. In other words, the upper-most two bits should be either binary 01 or 10 [14]. This is needed because HIP messages are distinguished from plain IPv6 traffic by defining two IPsec security policies for HITs (see 3.3.1).

In HIPL, the HIP associations are created as a side effect of resolver calls. In the opportunistic mode, the resolver would be queried with a peer host name that does not match any HIT in the */etc/hip/hosts* file or in the DNS. In this case, resolver would create a temporary HIT value and communicate it together with the peer's IPv6 address to the HIP module. The HIP module

would use the temporary value as the peer HIT and a fresh HADB entry with the state UNASSOCIATED would be created per every local HI.

The HIP association entries could be updated with the correct peer HIT value when the R1 message was received. Specifically, the HADB entries with temporary peer HITs should be removed and completely new ones should be created to replace them.

It could turn out to be difficult to implement the opportunistic base exchange transparently with the legacy API. An alternative solution would be to allow the opportunistic mode only through the native HIP API. Komu et al. propose that the native API's Endpoint Descriptors could be used to simplify the implementation [10].

7.4 Loopback

Currently the loopback does not work in HIPL. The implementation tries to create two distinct host association entries but fails because the HADB cannot contain two distinct entries with equal HIT pairs. Some modifications would be needed in the HIPL message processing in order to achieve the loopback functionality.

The terms initiator and responder that are used in the following discussion denote the same host. The communication takes place between two endpoints that are represented by differing local Host Identifiers.

The initiator of the base exchange first creates an association entry between a pair of local HITs, the initiating HIT and the destination HIT. The responder attempts to create a state and allocate a fresh HIP association when it receives the I2 message. Now, an association already exists when the responder tries to create a new one. To make the loopback function correctly, the I2 handling should be modified. The special loopback situation would be easiest to handle with only one host association entry since it is not possible to have two separate entries in the HADB with the same HIT pair. The implementation should be modified so that the special case of initiating the base exchange with a local HIT would be handled differently from the normal control flow.

7.5 User-specific Identifiers

The current implementation works as an enabler for user-specific identifiers but the actual deployment was not included in the scope of the thesis. If the user-specific identifiers are considered useful in the future, some simple scripts could be introduced to enable easy creation of the corresponding cryptographic keys for ordinary users. However, the actual usage scenarios of HIP are ultimately defined together with the deployment of real-life applications. For example, there is still no certainty whether the HIs are created on the local host or whether they are distributed to the hosts from an external location. The latter option could be feasible for a security-aware corporation with a centralized system administration.

7.6 Pre-assigned System Identifiers

Currently, the HIPL implementation resolves local HIs from the */etc/hip/* directory every time either *getendpointinfo* or *getaddrinfo* is called with NULL node name. HIs are communicated from user space to the HIPL kernel module per a resolver call. Furthermore, the current implementation exposes the private keys in */etc/hip* to all users because the resolver implementation requires that the key files must be all readable.

A more efficient solution could be to commit the HI assignment only once during the initialization of the kernel module. The resolver could be modified so that the user space processes would not search HIs from the local file system. The kernel would be queried for the related information instead. In this scheme, only system administrators with root privileges could reassign, delete or add new system identifiers. Normal user processes would not need to have even read access to the actual key files. The proposed scheme would be more secure than the current implementation since the identifiers were not unnecessarily exposed.

7.7 Discarding HIP Associations

Multiple identifiers enable a multitude of simultaneous associations. However, in the current implementation HIP associations, once created, are not ever removed. The HIP specification states that an implementation is free to drop a HIP association at any time based on its own policy [14]. Also the

HIPL implementation should include such a mechanism in order to optimize the memory consumption inside the kernel space.

It would be reasonable to use a timer and remove all unused associations after a given time period. The actual work could be committed by a dedicated kernel thread or a timed kernel task, for example.

7.8 Application Mobility

The ability to host multiple identities on a single networking node opens possibilities for application mobility. Koponen et al. describe the topic in a paper [11]. In the proposed architecture, a source entity delegates the usage rights of its HIP endpoint identity to the destination entity. As a result, the destination entity is authorized to represent the endpoint that was earlier located in the source host. The proposal suggests that the authorization is conducted by using delegation certificates.

Considering this thesis, the possibility to support multiple application specified identifiers within a host is certainly valuable if the application mobility functionality is implemented in the future by using HIPL. While a single local HI provides persistent transport layer connectivity for a mobile host, the support for multiple identifiers enables application mobility analogously.

7.9 Enforcing Explicit Bind

For the moment, the HIP APIs do not enforce an explicit bind related to the client sockets. Consequently, the HIP base exchange is initialized with an undetermined HI if the client application omits the bind call. In that case, the HIPL implementation selects the used local HI at random. One could consider whether this is appropriate or whether the HIP APIs should be redefined so that an explicit bind would be coerced also for the client sockets.

The rationale behind the legacy API is to enable HIP transparently so that the API appears like a standard socket API for the applications. Thus, the requirement for an explicit bind might turn out problematic concerning the API. On the other hand, the HIP native API could be easily redefined to enforce the binding for client sockets.

7.10 User space version of HIPL

Some of the functionalities that were added to the kernel space version of HIPL were not included in the user space implementation. In the user space version, the HIP socket handler resides in the kernel as usual but HIs are stored in the user space. This architecture poses problems in certain situations. For example, one should verify that the local HI really exists when a socket is tried to be bound to a HIT. In order to implement the verification process in the current scheme, a message should be sent back and forth between the kernel and user space. Some performance measurements could be conducted to decide whether this kind of approach would be appropriate. An alternative solution could be to replicate the HI data in the kernel space. In that case, memory consumption in the kernel space could be minimized by using HITs instead of full HIs.

Chapter 8

Conclusion

The Host Identity Protocol is a new and promising approach to facilitate mobility and multihoming by adding a new layer of names to the existing networking stack. The HIP specifications have not been completely stabilized and consequently the protocol is a target for a lot of ongoing research activities. More work is still needed to better understand the effects that HIP may have on the networking applications and on the Internet.

The consequences of hosting multiple HIP identifiers in a computing node have not been studied thoroughly. The purpose of this thesis was to enable multiple simultaneous Host Identifiers within a HIP enabled host. A functional proof-of-concept implementation was developed by enhancing the HIPL implementation of the HIP protocol.

The legacy HIPL implementation supported properly only one identifier at a time. The internal data structures of the implementation were modified to support a multitude of distinct identifiers. In addition, binding to local identifiers was enabled for both the plain IPv6 and native HIP sockets. Further, endpoint resolution for multiple identifiers was provided by modifying the resolver library implementation.

Regarding HIPL, identifiers are divided to pre-assigned *system identifiers* and so called *application specified identifiers*. An access control mechanism was implemented concerning especially the latter ones. An Endpoint Descriptor (ED) works as a reference to an application specified HI and serves as the basis for the related access control.

The support for multiple simultaneous HIs within a host opens new possibilities for HI usage and further research. An analysis concerning the possible implications from hosting multiple HIs was presented and potential items for

future work were identified.

The implementation effort brought up a number of ideas for further development. For instance, the internal message processing needs to be modified in order to provide some missing functionalities such as the loopback interface and the opportunistic mode. In addition, some changes might be needed in the HIP API syntax.

The implementation allows to restrict access to a local HI based on the UID value of the binding process. Consequently, HIs can be assigned to individual users. User-specific identifiers open further interesting prospects. For instance, a HIP enabled firewall could provide per-user rules.

A natural next step would be to exploit the improved functionalities of HIPL in some real-life applications. A well-known free software would be an ideal candidate to draw attention in the open source community.

Bibliography

[1] Balakrishnan, H., Lakshminarayanan, K., Ratnasamy, S., Shenker, S., Stoica, I., and Walfish, M. A Layered Naming Architecture for the Internet. In *ACM SIGCOMM 2004* (Portland, OR, September 2004).

[2] Candolin, C., Komu, M., Kousa, M., and Lundberg, J. An Implementation of HIP for Linux. In *Proc. of the Linux Symposium* (July 2003).

[3] Clark, D. D., Sollins, K., Wroclawski, J., and Faber, T. Addressing Reality: an Architectural Response to Real-world Demands on the Evolving Internet. *SIGCOMM Comput. Commun. Rev. 33*, 4 (2003), 247–257.

[4] Gilligan, R., Thomson, S., Bound, J., McCann, J., and Stevens, W. RFC 3493: Basic Socket Interface Extensions for IPv6. Tech. rep., IETF, Feb. 2003.

[5] Herbert, T. F. *The Linux TCP/IP Stack: Networking for Embedded Systems.* Charles River Media, Inc., 2004.

[6] Jokela, P., Moskowitz, R., and Nikander, P. Using ESP transport format with HIP. Internet draft, IETF, Feb 2005. Work in progress. Expires August 12, 2005.

[7] Kent, S., and Atkinson, R. RFC 2401 - Security Architecture for the Internet Protocol. Tech. rep., IETF, Nov. 1998.

[8] Komu, M. Application Programming Interfaces for the Host Identity Protocol. Master's thesis, Helsinki University of Technology, 2004.

[9] Komu, M. Native Application Programming Interfaces for the Host Identity Protocol. Internet draft, IETF, March 2005. Work in progress. Expires August 30, 2005.

[10] Komu, M., Tarkoma, S., Kangasharju, J., and Gurtov, A. Applying a Cryptographic Namespace to Applications. Unpublished manuscript., 2005.

[11] Koponen, T., Gurtov, A., and Nikander, P. Application mobility with Host Identity Protocol. In *Proc. of NDSS Wireless and Security Workshop* (Feb. 2005), Internet Society.

[12] Love, R. *Linux Kernel Development, Second Edition.* Novell Press, 2005.

[13] Moskowitz, R., and Nikander, P. Host Identity Protocol Architecture. Internet draft, IETF, Jan 2005. Work in progress. Expires August, 2005.

[14] Moskowitz, R., Nikander, P., Jokela, P., and Henderson, T. Host Identity Protocol. Internet draft, IETF, Feb 2005. Work in progress. Expires December 25, 2005.

[15] Nikander, P., and Laganier, J. Host Identity Protocol (HIP) Domain Name System (DNS) Extensions. Internet draft, IETF, Feb 2005. Work in progress. Expires August 21, 2005.

[16] Nikander, P., Ylitalo, J., and Wall, J. Integrating security, mobility, and multi-homing in a HIP way. In *Proc. of Network and Distributed Systems Security Symposium (NDSS'03)* (Feb. 2003).

[17] The USAGI Project. Linux IPv6 Development Project.

[18] Ylitalo, J., Jokela, P., Wall, J., and Nikander, P. Endpoint Identifiers in Secure Multi-Homed Mobility. In *OPODIS'02, Reims, France, December 11-13* (December 2002), Universite de Reims Champagne-Ardenne, pp. 17–28.

[19] Ylitalo, J., and Nikander, P. Blind: A Complete Identity Protection Framework for End-points. In *Twelfth International Workshop on Security Protocols* (April 2004), Cambridge, England.

Appendix A

Functional Tests

A.1 Responder Test

```
/*
 * multiple_r1_test.c
 *
 * Tests the proper handling of received I1
 * messages on the server side. Use this test client
 * with conntest-server running on the peer host.
 *
 * Resolves the peer HITs from /etc/hip/hosts by
 * using the legacy API getaddrinfo resolver routine
 * and initiates the HIP base exhange with each
 * of the resolved peer endpoints/HITs in a row.
 *
 * Note: If multiple identifiers are wanted to be
 * tested on the server side, the client end's
 * /etc/hip/hosts file should contain multiple
 * HITs associated with the peer host entry.
 * (<hit1> <hit2> <hitX> <peer host name>)
 *
 * Usage: ./multiple_r1_test peername port
 */

#if HAVE_CONFIG_H
#include <config.h>
#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#include <time.h>
#include "libinet6/debug.h"

int main(int argc,char *argv[]) {
  struct timeval stats_before, stats_after;
  unsigned long stats_diff_sec, stats_diff_usec;
  char *proto_name, *peer_port_name, *peer_name;
  char sentdata[1], receiveddata[1];
  int recvnum, sendnum, datasent = 0, datareceived = 0;
  int proto = 0, gai_err, sock = 0, socktype;
  char i = 'a';
  struct addrinfo hints, *res = NULL, *ai;

  if (argc != 3) {
    fprintf(stderr, "Usage: %s host port\n", argv[0]);
    exit(1);
  }

  peer_name = argv[1];
  peer_port_name = argv[2];

  proto = IPPROTO_TCP;
  socktype = SOCK_STREAM;

  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_flags = AI_HIP;
  hints.ai_family = AF_INET6;
  hints.ai_socktype = socktype;
  hints.ai_protocol = proto;
```

```
gai_err = getaddrinfo(peer_name, peer_port_name,
                      &hints, &res);
if (gai_err) {
  printf("GAI ERROR %d: %s\n", gai_err,
 gai_strerror(gai_err));
  return(1);
}

for(ai = res; ai != NULL; ai = ai->ai_next, i++) {
  sentdata[0] = i;
  struct sockaddr_in6 *sin6 =
    (struct sockaddr_in6 *) ai->ai_addr;
  char addr_str[INET6_ADDRSTRLEN];

  HIP_ASSERT(ai->ai_family == AF_INET6);

  sock = socket(AF_INET6, SOCK_STREAM, 0);
  if (sock < 0) {
    sock = 0;
    printf("socket creation failed\n");
    goto out_err;
  }

  if (!inet_ntop(AF_INET6,
                 (char *) &sin6->sin6_addr,
                 addr_str, sizeof(addr_str))) {
    perror("inet_ntop\n");
    goto out_err;
  }

  printf("Trying to connect to %s\n", addr_str);

  gettimeofday(&stats_before, NULL);

  if (connect(sock, ai->ai_addr,
              sizeof(struct sockaddr_in6)) < 0) {
    close(sock);
    sock = 0;
    printf("error: trying next\n");
    continue;
```

```
    }

    gettimeofday(&stats_after, NULL);
    stats_diff_sec  = (stats_after.tv_sec -
       stats_before.tv_sec) * 1000000;
    stats_diff_usec = stats_after.tv_usec -
      stats_before.tv_usec;

    printf("connect took %.3f sec\n",
           (stats_diff_sec+stats_diff_usec) / 1000000.0);

    /* send and receive data */

    sendnum = send(sock, sentdata, 1, 0);
    if (sendnum <= 0) {
      printf("Sending failed\n");
      goto out_err;
    }
    recvnum = recv(sock, receiveddata, 1, 0);
    if (recvnum <= 0) {
      printf("Receive failed\n");
      goto out_err;
    }

    if (!memcmp(sentdata, receiveddata, 1)) {
      printf("Data transfer OK\n");
    } else {
      printf("Data transfer failed\n");
    }
    close(sock);
  }

 out_err:

  if (res)
    freeaddrinfo(res);
  if (sock)
    close(sock);
  return 0;

}
```

A.2 Multiple Identifiers and HIP Associations Test

```
/*
 * multiple_id_bind_test.c
 *
 * Tests multiple local identifiers and multiple HIP
 * associations. Resolves local system identifiers
 * from /etc/hip/ directory and initiates the HIP
 * base exchange with a remote host by using each of
 * the local HIs in a row.
 *
 * Note: Use this test with conntest-server running on
 * the peer machine.
 *
 * Usage: ./multiple_id_bind_test peername port
 *
 */

#if HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#include <time.h>
#include <arpa/inet.h>
#include <net/if.h>
#include "libinet6/debug.h"

int main(int argc,char *argv[]) {
```

```
struct endpointinfo hints, *res = NULL,
  *res1 = NULL, *epinfo1 = NULL;
struct sockaddr_eid *my_eid;
struct timeval stats_before, stats_after;
unsigned long stats_diff_sec, stats_diff_usec;
char sentdata[1], receiveddata[1];
char *proto_name, *peer_port_name, *peer_name;
int recvnum, sendnum, datasent = 0, datareceived = 0;
char i = 'a';
int err = 0, proto = 0, sockfd = 0, socktype;
se_family_t endpoint_family;

if (argc != 3) {
  HIP_ERROR("Usage: %s host port\n", argv[0]);
  err = 1;
  goto out;
}

peer_name = argv[1];
peer_port_name = argv[2];
endpoint_family = PF_HIP;

/* Set transport protocol */
proto = IPPROTO_TCP;
socktype = SOCK_STREAM;

/* Set up endpoint lookup information  */
memset(&hints, 0, sizeof(struct endpointinfo));
hints.ei_socktype = socktype;
hints.ei_family = endpoint_family;

/* Lookup local endpoints. */
err = getendpointinfo(NULL, peer_port_name, &hints, &res1);
if (err) {
  HIP_ERROR("getendpointinfo failed (%d): %s\n",
            err, gepi_strerror(err));
  goto out;
}

epinfo1 = res1;
```

```
/* Lookup peer endpoint. */
err = getendpointinfo(peer_name, peer_port_name,
                      &hints, &res);
if (err) {
  HIP_ERROR("getendpointinfo failed (%d): %s\n", err,
            gepi_strerror(err));
  goto out;
}

while(epinfo1) {

  sockfd = socket(endpoint_family, socktype, 0);
  if (sockfd == -1) {
    HIP_ERROR("creation of socket failed\n");
    err = 1;
    goto out;
  }

  /* Bind a local HI to the socket */
  err = bind(sockfd, epinfo1->ei_endpoint,
             epinfo1->ei_endpointlen);
  if (err == -1) {
    HIP_PERROR("bind failed");
    goto out;
  }

  gettimeofday(&stats_before, NULL);

  err = connect(sockfd, res->ei_endpoint,
                res->ei_endpointlen);
  if (err == -1) {
    HIP_PERROR("connect");
    printf ("** Error connecting socket,
            errno = %d **\n",errno);
    goto out;
  }

  gettimeofday(&stats_after, NULL);
  stats_diff_sec  = (stats_after.tv_sec -
                     stats_before.tv_sec) *
                    1000000;
```

```
    stats_diff_usec = stats_after.tv_usec -
                      stats_before.tv_usec;

    HIP_DEBUG("connect took %.10f sec\n",
              (stats_diff_sec + stats_diff_usec) /
              1000000.0);

    /* send and receive data */
    sentdata[0] = i;

    sendnum = send(sockfd, sentdata, 1, 0);
    if (sendnum <= 0) {
      printf("Sending failed\n");
      goto out;
    }

    recvnum = recv(sockfd, receiveddata, 1, 0);
    if (recvnum <= 0) {
      printf("Receive failed\n");
      goto out;
    }

    if (!memcmp(sentdata, receiveddata, 1)) {
      printf("Data transfer OK\n");
    } else {
      printf("Data trasnfer failed\n");
    }

    shutdown(sockfd, SHUT_RDWR);
    epinfo1 = epinfo1->ei_next;
    i++;

  }

out:

  if (res)
     free_endpointinfo(res);
  if (res1)
     free_endpointinfo(res1);
```

```
  return err;
}
```

A.3 Test Server

```
/*
 * conntest-server.c
 *
 * Receives data from a client and sends it back (echo server).
 *
 */

#if HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <unistd.h>
#include <netdb.h>
#include <signal.h>
#include "libinet6/debug.h"

int create_serversocket(int proto, int port) {
  int fd, on = 1;
  struct sockaddr_in6 addr;

  if (proto == IPPROTO_TCP) {
    fd = socket(AF_INET6, SOCK_STREAM, 0);
  } else {
    fd = socket(AF_INET6, SOCK_DGRAM, 0);
  }
  if (fd < 0) {
```

```
    perror("socket");
    exit(1);
  }

  setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

  bzero(&addr, sizeof(addr));
  addr.sin6_family = AF_INET6;
  addr.sin6_port = htons(port);
  addr.sin6_addr = in6addr_any;
  addr.sin6_flowinfo = 0;

  if (bind(fd, (struct sockaddr *)&addr,
   sizeof(struct sockaddr_in6)) < 0) {
    perror("bind");
    close(fd);
    exit(1);
  }

  if (proto == IPPROTO_TCP) {
    if (listen(fd, 1) < 0) {
      perror("listen");
      close(fd);
      exit(1);
    }
  }

  return(fd);
}


int main(int argc,char *argv[]) {

  int serversock;
  int peer;
  unsigned int peerlen;
  struct sockaddr_in6 peeraddr;
  char data[IP_MAXPACKET];
  int recvnum, sendnum;
  char addrstr[INET6_ADDRSTRLEN];
```

```
int port;
int proto;

if (argc != 3) {
  fprintf(stderr, "Usage: %s tcp|udp port\n", argv[0]);
  exit(1);
}

if (strcmp(argv[1], "tcp") == 0) {
  proto = IPPROTO_TCP;
} else if (strcmp(argv[1], "udp") == 0) {
  proto = IPPROTO_UDP;
} else {
  fprintf(stderr, "error: protonum != tcp|udp\n");
  exit(1);
}

port = atoi(argv[2]);
if (port <= 0 || port >= 65535) {
  fprintf(stderr, "error: port < 0 || port > 65535\n");
  exit(1);
}
serversock = create_serversocket(proto, port);

peerlen = sizeof(struct sockaddr_in6);

while(1) {

  if (proto == IPPROTO_TCP) {
    peer = accept(serversock,
                  (struct sockaddr *)&peeraddr,
                  &peerlen);
    if (peer < 0) {
      perror("accept");
      exit(2);
    }

    while((recvnum = recv(peer, data,
  sizeof(data), 0)) > 0 ) {
      data[recvnum] = '\0';
      printf("%s", data);
```

```
      fflush(stdout);
      if (recvnum == 0) {
        close(peer);
        break;
      }

      /* send reply */
      sendnum = send(peer, data, recvnum, 0);
      if (sendnum < 0) {
        perror("send");
        exit(2);
      }
    }
  } else { /* UDP */
    peerlen = sizeof(struct sockaddr_in6);
    peer = serversock;
    while((recvnum = recvfrom(peer, data,
                              sizeof(data),
                              0,
                              (struct sockaddr *)&peeraddr,
                              &peerlen)) > 0 ) {

      data[recvnum] = '\0';
      fprintf(stderr,"%s", data);
      fflush(stdout);
      if (recvnum == 0) {
        close(peer);
        break;
      }

      /* send reply */
      sendnum = sendto(peer,
                       data,
                       recvnum,
                       0,
                       (struct sockaddr *)&peeraddr,
                       peerlen);
      if (sendnum < 0) {
        perror("send");
        exit(2);
      }
```

```
      }
    }
  }

  close(peer);
  close(serversock);
  return(0);
}
```

A.4 Resolver Tests

A.4.1 getendpointinfo

```
/*
 * getendpointinfo.c
 *
 * getendpointinfo test program
 * Accepts peer host name and service name as input arguments.
 *
 * getendpointinfo() function is tested both for a server
 * application use (with NULL node name) and also for a
 * typical client application use (with peer host name as
 * the nodename parameter).
 *
 */

#if HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdlib.h>

int main(int argc, char **argv) {
```

```
int a;
struct endpointinfo hints, *res = NULL;
struct endpointinfo *ai = NULL;

if (argc != 3) {
  printf("%s nodename servname\n", argv[0]);
  exit(2);
}

hints.ei_family = PF_HIP;
hints.ei_socktype = SOCK_STREAM;
hints.ei_protocol = 0;

printf("\ngetendpointinfo(NULL, %s, &hints, &res)",
       argv[2]);
a = getendpointinfo(NULL, argv[2], &hints, &res);

if (a) {
  printf("ERROR: Resolving of peer identifiers failed (%d)\n", a);
  return(1);
}

for(ai = res; ai != NULL; ai = ai->ei_next) {
  printf("ei_flags=%d ei_family=%d ei_socktype=%d ei_protocol=%d\n",
 ai->ei_flags, ai->ei_family, ai->ei_socktype, ai->ei_protocol);
  printf("ei_endpoint_len=%d ei_canonname=%s\n",
 ai->ei_endpointlen, ai->ei_canonname);

  if (ai->ei_family == PF_HIP) {
    struct sockaddr_eid *s = (struct sockaddr_eid *)ai->ei_endpoint;
    int i = 0;

    printf("PF_HIP\t eid_family=%d\n", s->eid_family);
    printf("PF_HIP\t eid_port=%d\n", ntohs(s->eid_port));
    printf("PF_HIP\t eid_val=%d\n", ntohs(s->eid_val));

  }
}

free_endpointinfo(res);
```

```
  printf("\ngetendpointinfo(%s, %s, hints, res)",
         argv[1],argv[2]);
  a = getendpointinfo(argv[1], argv[2], &hints, &res);

  if (a != 0) {
    printf("*** ERROR: %s ***\n", gai_strerror(a));
    return(1);
  }

  for(ai = res; ai != NULL; ai = ai->ei_next) {
    printf("ei_flags=%d ei_family=%d ei_socktype=%d ei_protocol=%d\n",
           ai->ei_flags, ai->ei_family, ai->ei_socktype,
           ai->ei_protocol);
    printf("ei_endpoint_len=%d ei_canonname=%s\n",
           ai->ei_endpointlen, ai->ei_canonname);

    if (ai->ei_family == PF_HIP) {
      struct sockaddr_eid *s = (struct sockaddr_eid *)ai->ei_endpoint;
      int i = 0;

      printf("PF_HIP\t eid_family=%d\n", s->eid_family);
      printf("PF_HIP\t eid_port=%d\n", ntohs(s->eid_port));
      printf("PF_HIP\t eid_val=%d\n", ntohs(s->eid_val));
    }
  }

  free_endpointinfo(res);
  return(0);
}
```

A.4.2 getaddrinfo

```
/*
 * getaddrinfo.c
 *
 * getaddrinfo test program
 * Accepts peer host name and service name as input arguments.
 *
 * getaddrinfo() function is tested both for a server
```

```
 * application use (with NULL node parameter and
 * AI_PASSIVE flag set) and also for a typical
 * client application use (with peer host name as
 * the nodename parameter).
 *
 */

#if HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdlib.h>

int main(int argc, char **argv) {

  int a;
  struct addrinfo hints;
  struct addrinfo *res;
  struct addrinfo *ai;

  if (argc != 3) {
    printf("%s nodename servname\n", argv[0]);
    exit(2);
  }

  hints.ai_flags = AI_HIP | AI_PASSIVE;
  hints.ai_family = AF_INET6;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_protocol = 0;

  printf("\ngetaddrinfo(NULL, %s, hints, res)", argv[2]);
  a = getaddrinfo(NULL, argv[2], &hints, &res);

  if (a != 0) {
    printf("*** ERROR: %s ***\n", gai_strerror(a));
```

```
    return(1);
  }

  for(ai = res; ai != NULL; ai = ai->ai_next) {
    printf("ai_flags=%d ai_family=%d ai_socktype=%d ai_protocol=%d\n",
           ai->ai_flags, ai->ai_family, ai->ai_socktype,
           ai->ai_protocol);
    printf("ai_addrlen=%d ai_canonname=%s\n",
           ai->ai_addrlen, ai->ai_canonname);

    if (ai->ai_family == AF_INET6) {
      struct sockaddr_in6 *s = (struct sockaddr_in6 *)ai->ai_addr;
      int i = 0;

      printf("AF_INET6\t sin6_family=%d\n", s->sin6_family);
      printf("AF_INET6\t sin6_port=%d\n", ntohs(s->sin6_port));
      printf("AF_INET6\t sin6_flowinfo=%lu\n",
             (long unsigned int)s->sin6_flowinfo);
      printf("AF_INET6\t sin6_scope_id=%lu\n",
             (long unsigned int)s->sin6_scope_id);
      printf("AF_INET6\t sin6_addr=0x");
      for (i = 0; i < 16; i++)
        printf("%02x ", (unsigned char)
               (s->sin6_addr.in6_u.u6_addr8[i]));
      printf("\n");
    } else if (ai->ai_family == AF_INET) {
      struct sockaddr_in *s = (struct sockaddr_in *)ai->ai_addr;
      printf("AF_INET\t in_addr=0x%lx (%s)\n",
     (long unsigned int) ntohl(s->sin_addr.s_addr),
     inet_ntoa(s->sin_addr));
    }
  }

  freeaddrinfo(res);

  hints.ai_flags = AI_HIP;
  hints.ai_family = AF_INET6;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_protocol = 0;

  printf("\ngetaddrinfo(%s, %s, hints, res)",
```

```
        argv[1], argv[2]);
a = getaddrinfo(argv[1], argv[2], &hints, &res);

if (a != 0) {
  printf("*** ERROR: %s ***\n", gai_strerror(a));
  return(1);
}

for(ai = res; ai != NULL; ai = ai->ai_next) {
  printf("ai_flags=%d ai_family=%d ai_socktype=%d ai_protocol=%d\n",
         ai->ai_flags, ai->ai_family, ai->ai_socktype,
         ai->ai_protocol);
  printf("ai_addrlen=%d ai_canonname=%s\n",
 ai->ai_addrlen, ai->ai_canonname);

  if (ai->ai_family == AF_INET6) {
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)ai->ai_addr;
    int i = 0;

    printf("AF_INET6\t sin6_family=%d\n", s->sin6_family);
    printf("AF_INET6\t sin6_port=%d\n", ntohs(s->sin6_port));
    printf("AF_INET6\t sin6_flowinfo=%lu\n",
           (long unsigned int)s->sin6_flowinfo);
    printf("AF_INET6\t sin6_scope_id=%lu\n",
           (long unsigned int)s->sin6_scope_id);
    printf("AF_INET6\t sin6_addr=0x");
    for (i = 0; i < 16; i++)
      printf("%02x ", (unsigned char)
             (s->sin6_addr.in6_u.u6_addr8[i]));
    printf("\n");
  } else if (ai->ai_family == AF_INET) {
    struct sockaddr_in *s = (struct sockaddr_in *)ai->ai_addr;
    printf("AF_INET\t in_addr=0x%lx (%s)\n",
   (long unsigned int) ntohl(s->sin_addr.s_addr),
   inet_ntoa(s->sin_addr));
  }
}


freeaddrinfo(res);
return(0);
```

```
}
```

A.5 HI Binding Test

```
/*
 * app_specified_key_test.c
 *
 * Loads a PEM formatted key from the specified file.
 * A server socket is bound to the ED that is
 * constructed from the loaded key. Next, a simple
 * echo server is started. The server listens
 * at the specified port and accepts connections
 * destined to the HIT that is calculated from the
 * loaded HI.
 *
 * The flag parameter indicates whether the HI binding
 * is allowed to be used by other processes or not.
 *
 * Usage: ./app_specified_key_test port key_file flag
 * (no flag = 0, HIP_HI_REUSE_UID = 1,
 *  HIP_HI_REUSE_GID = 2, HIP_HI_REUSE_ANY = 3)
 *
 */

#if HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
```

```
#include <time.h>
#include <arpa/inet.h>
#include <net/if.h>
#include "libinet6/debug.h"

int main(int argc,char *argv[]) {
  struct endpointinfo *res = NULL;
  struct sockaddr_eid *my_eid = NULL, peer_eid;
  char *port_name, receiveddata[IP_MAXPACKET];
  int err = 0, sockfd = 0, serversock = 0, socktype,
    recvnum, sendnum, flag;
  se_family_t endpoint_family;
  char *user_key_base;
  struct endpoint *endpoint;
  socklen_t peer_eid_len;

  if (argc != 4) {
    printf("Usage: %s port key_file flag(0-3)\n", argv[0]);
    err = 1;
    goto out;
  }

  switch (atoi(argv[3])) {
  case 0:
    flag = 0;
    break;
  case 1:
    flag = HIP_HI_REUSE_UID;
    break;
  case 2:
    flag = HIP_HI_REUSE_GID;
    break;
  case 3:
    flag = HIP_HI_REUSE_ANY;
    break;
  default:
    printf("Flag: 1-3\n");
    goto out;
  }

  port_name = argv[1];
```

```
user_key_base = argv[2];
endpoint_family = PF_HIP;
socktype = SOCK_STREAM;

serversock = socket(endpoint_family, socktype, 0);
if (sockfd == -1) {
  printf("creation of socket failed\n");
  err = 1;
  goto out;
}

err = load_hip_endpoint_pem(user_key_base, &endpoint);
if (err) {
  printf("Failed to load user HIP key %s\n", user_key_base);
  goto out;
}

my_eid = getlocaled(endpoint, port_name, NULL, NULL, flag);
if (err) {
  printf("Failed to set up my EID (%d)\n", err);
  err = 1;
  goto out;
}

err = bind(serversock, (struct sockaddr *)my_eid,
   sizeof(struct sockaddr_eid));
if (err) {
  HIP_PERROR("bind failed\n");
  goto out;
}

if (listen(serversock, 1) < 0) {
  printf("listen");
  err = 1;
  goto out;
}

while(1) {

  sockfd = accept(serversock, (struct sockaddr *) &peer_eid,
  &peer_eid_len);
```

```
    if (sockfd < 0) {
      printf("accept failed\n");
      err = 1;
      goto out;
    }

    while((recvnum = recv(sockfd, receiveddata,
                          sizeof(receiveddata), 0)) > 0 ) {
      receiveddata[recvnum] = '\0';
      printf("%s", receiveddata);
      fflush(stdout);
      if (recvnum == 0) {
        break;
      }

      sendnum = send(sockfd, receiveddata, recvnum, 0);
      if (sendnum < 0) {
        printf("send failed");
        err = 1;
        goto out;
      }
    }
  }

 out:

  if (sockfd)
    close(sockfd);
  if (serversock)
    close(serversock);
  if (res)
    free_endpointinfo(res);
  if(my_eid)
    free(my_eid);

  return err;
}
```