Host Identity Protocol
Internet-Draft
Intended status: Informational
Expires: May 22, 2008

M. Komu
Helsinki Institute for Information Technology

November 19, 2007

Native Application Programming Interfaces (APIs) for Host Identity Protocol (HIP) draft-ietf-hip-native-api-03

Status of this Memo

Copyright Notice

Abstract

This document defines extensions to the current sockets API for Host Identity Protocol (HIP). The extensions focus on the initial discovery of public-key based identifiers. Using the extensions, the application can verify that the identifier is a Host Identity Tag

(HIT) and it can require the system resolver to return only HITs from DNS. The application can also to explicitly allow more relaxed security models where the communication can be non-HIP based in the absence of a peer identifiers, or that the application allows peer identity to be discovered after initial contact directly with the peer.

Table of Contents

1. Terminology

The terms used in this document are summarized in Table 1.

Term	Explanation
HIP	Host Identity Protocol
HIT	Host Identity Tag, a 100-bit hash of a public key with a 28 bit prefix
LSI	Local Scope Identifier, a local, 32-bit descriptor for a given public key.
Locator	Routable IPv4 or IPv6 address used at the lower layers

Table 1

2. Introduction

The Host Identity Protocol (HIP) [RFC4423] proposes a new cryptographic namespace by separating the roles of end-point identifiers and locators by introducing a new namespace to the TCP/IP stack. SHIM6 [I-D.ietf-shim6-proto] is another protocol based on identity-locator split. Note that the Application Programming Interfaces (APIs) specified in this document are specific to HIP. However, the APIs here have been designed keeping generality in mind as much as possible so as not to preclude its use with other protocols. The use of these APIs with other protocols is, nevertheless, for further study.

Applications can observe the HIP layer and its identifiers in the networking stacks with varying degrees of visibility. [I-D.henderson-hip-applications] discusses the lowest levels of visibility in which applications are completely unaware of the underlying HIP layer. Such HIP-unaware applications use HIP-based

identifiers, such as LSIs or HITs, instead of IPv4 or IPv6 addresses and cannot observe the identifier-locator bindings.

This document defines C-based sockets API extensions for handling HIP-based identifiers explicitly in HIP-aware applications. It is up to the applications, or a high-level programming languages or libraries, to manage the identifiers. The extensions in this document are mainly related to the initial discovery of the identifiers, i.e., DNS resolution step.

The API extensions introduce a new address family, AF_HIP, and a new socket address structure for sockets using Host Identity Tags (HITs)

explicitly. PF_HIP is used as an alias for AF_HIP in this document because the distinction between PF and AF has been lost in the practice.

Some applications may accept incoming communications from any identifier. Other applications may initiate outgoing communications without knowledge of the peer identifier in Opportunistic Mode [I-D.ietf-hip-base] by just relying on a peer locator. This document describes how to address both situations using "wildcards" as described later in this document.

There are two related API documents. Multihoming and explicit locator-handling related APIs are defined in [I-D.ietf-shim6-multihome-shim-api]. IPsec related policy attributes and channel bindings APIs are defined in [I-D.ietf-btns-c-api]. The extensions defined in this document can be used independently of the two mentioned related API documents.

To recap, the extensions in this document have two goals. The first goal is to allow HIP-aware applications to resolve HITs explicitly. The second goal is that applications can explicitly accept communications with unknown peer identifiers.

3. Design Model

In this section, the native HIP APIs is described from a design point of view. We first describe the namespace model and conclude the discussion with a description of the resolver model.

3.1. Namespace Model

The namespace model is shown in Table 2 from HIP point of view. The namespace identifiers are described in this section.

Layer	Identifier
User Interface	Relative hostname or FQDN
Application Layer	HIT, port and protocol
Transport Layer	HIT, port
HIP Layer	HIT or HI
Network Layer	Locator

Table 2

User interfaces input human-readable names and translate them to

machine-readable names. In native APIs for HIP, the machine readable names are HITs. The HITs are present at the application layer, and transport-layer pseudo checksums are based on HITs. The HIP layer transforms the HITs to locators for the network layer and vice versa.

3.2. Interaction with the Resolver

Before an application can establish network communications with the entity named by a given FQDN or relative host name, the application must translate the name into the corresponding identifier(s). DNS based hostname-to-identifier translation is illustrated in Figure 1. The application calls the resolver (step a.) to resolve an FQDN (step b.). The DNS server responds with a list of HITs and a set of locators (step c.). Optionally (in step d.), the resolver caches the HIT to locator mapping to the HIP module. The resolver returns the HITs to the application in step e. Finally, the application selects one HIT and uses it in a socket call such as connect() in step e.

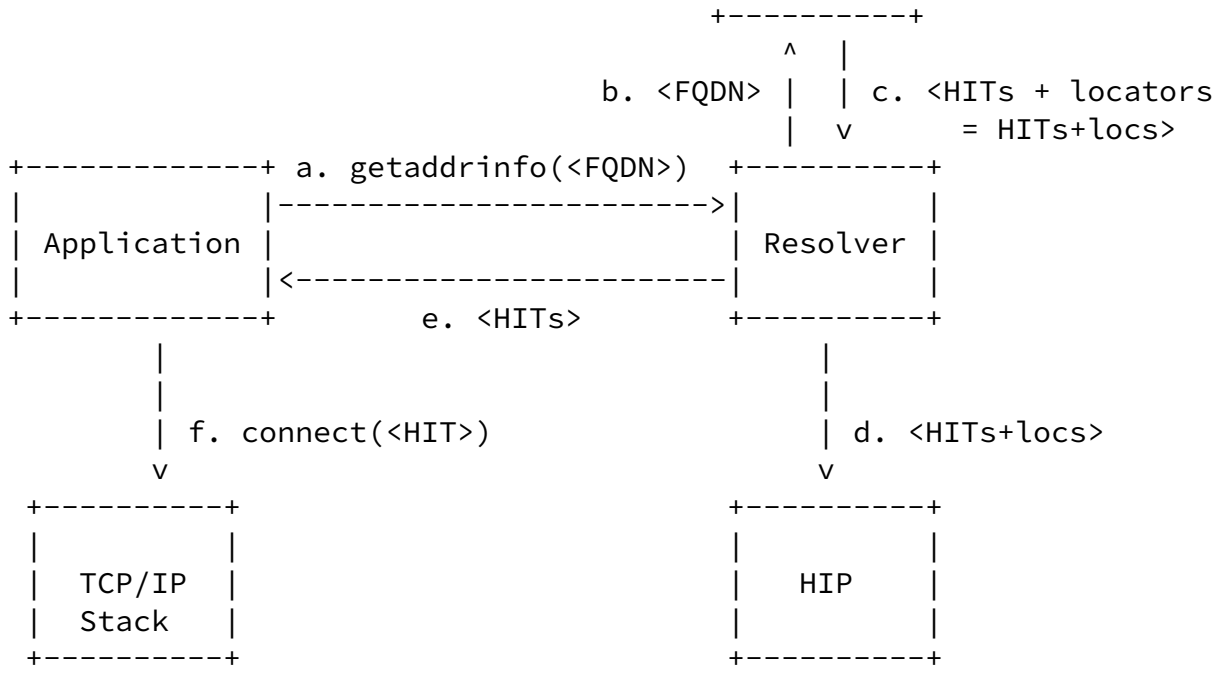

Figure 1

In practice, the resolver functionality can be implemented in different ways. For example, it may be implemented in existing resolver libraries or as a DNS proxy.

The extensions in this document focus on the use of the resolver to

map host names to HITs and locators in HIP-aware applications. The resolver associates implicitly the the HIT with the locator(s). However, it is possible that an application operates directly with a peer HIT without interacting with the resolver. In such a case, the application may resort to the system to map the peer HIT to an IP address. Alternatively, the application can explicitly map the HIT to an IP address as specified in [I-D.ietf-shim6-multihome-shim-api]. Both of these two approaches may be more prone to errors than the use resolver with host names. Hence, HIP-aware applications should prefer to use the resolver with host names.

4. API Syntax and Semantics

In this section, we describe the native HIP APIs using the syntax of

the C programming language. We limit the description to the interfaces and data structures that are either modified or completely new, because the native HIP APIs are otherwise identical to the sockets API [POSIX].

4.1. Socket Family and Address Structure Extensions

The sockets API extensions define a new protocol family, PF_HIP, and a new address family, AF_HIP. The AF_HIP and PF_HIP are aliases to each other. The use of the PF_HIP constant is mandatory with the socket() function when application uses the native HIP APIs. The application gives the PF_HIP constant as the first argument (domain) to the socket() function. The system returns EPFNOSUPPORT in the socket call when it does not support HIP.

The application can also use the new PF_HIP family to detect HIP support in the local host. Namely, the application creates a socket by calling socket() function with the first argument (domain) as PF_HIP. The system returns a positive integer representing a socket descriptor when the system supports HIP. Otherwise, the system returns -1 and sets errno to EAFNOSUPPORT.

A HIT is contained in a sockaddr_hip structure, which is shown in Figure 2. The family of the socket, ship_family, is set to PF_HIP. The port number ship_port is two octets and the sins_hit is four octets. The HIT value is an IPv6 address and it is stored in network byte order.


```
#include <netinet/in.h>

typedef struct in6_addr hip_hit_t;

struct sockaddr_hip {
        uint8_t        ship_len;
        uint8_t        ship_family;
        uint16_t       ship_port;
```

```
                uint64_t        ship_flags;
                hip_hit_t       ship_hit;
                uint8_t         reserved[16];
        }
```

Figure 2

The application usually sets the ship_hit field using the resolver. However, the application can use three special wildcard macros to set a value directly into the ship_hit field. The macros are HIP_HIT_ANY, HIP_HIT_ANY_PUB and HIP_HIT_ANY_ANON. They denote a HIT value associated with a wildcard HIT of any, public, or anonymous type. The HIP_HIT_ANY means HIP_HIT_ANY_PUB or HIP_HIT_ANY_ANON. The anonymous identifiers refer to the use anonymous identifiers as specified in [RFC4423]. The system may designate anonymous identifiers as meta data associated with a HIT regarding whether it has been published or not, but that from the HIP protocol perspective, there is no difference in the classes of HITs.

The application can use the macro HIP_IS_IPV6_ADDR_ANON_HIT to verify whether a HIT is anonymous or public. The macro inputs a pointer to a hip_hit_t structure and returns an integer (int) set to one when the corresponding HIT is anonymous and zero when it is public. The macro returns -1 when the anonymity status is not available.

The application can use the HIP_HIT_ macros to accept incoming communications to all of the HITs of the local host. Incoming communications refers here to the functions such as bind(), recvfrom() and recvmsg(). The HIP_HIT_ macros correspond to the sockets API macros INADDR_ANY and IN6ADDR_ANY_INIT, but they are applicable at the HIP layer. After initial contact with the peer, the local and peer HITs can be discovered using getsockname() and getpeername() calls for connection oriented sockets.

The application also uses the the HIP_HIT_ANY macro in ship_hit field to establish outgoing communications in Opportunistic mode [I-D.ietf-hip-base], when the application knows the remote peer locator but not the HIT. Outgoing communications refers here to the use of functions such as connect(), sendto() and sendmsg(). After initial contact with the peer, the application discovers local and

peer HITs using getsockname() and getpeername() calls when it is

using connection-oriented sockets.

The HIP_HIT_ANY_ macros also allow non-ORCHID based communications. To distinguish between ORCHID [RFC4843] and non-ORCHID-based communications in the case of the HIP_HIT_ANY_ macros, the application calls getsockname() and getpeername() to discover the actual identifiers used for the communications and verifies orchid prefix with HIP_IS_IPV6_ADDR_ORCHID macro. The macro inputs a pointer to an in6_addr structure and returns 1 when the address has orchid prefix and 0 otherwise. Alternatively, the application can set the flag HIP_FLAG_ONLY_ORCHID in ship_flags to allow only ORCHID-based communications.

Applications can also implement access control using the HITs. In such a case, the application can compare two HITs using memcmp() or similar function. It should be noticed that different connection attempts between the same two hosts can result in different HITs because a host is allowed to have multiple HITs.

4.2. Resolver Extensions

The HIP APIs introduces a new addrinfo flag, AI_HIP, to be used by application to query for both HIT and locator information via the getaddrinfo() resolver function [RFC3493]. The getaddrinfo() function uses a data structure used for both input to and output from the resolver. The data structure is illustrated in Figure 3.

```
#include <netdb.h>

struct addrinfo {
    int    ai_flags;          /* e.g. AI_HIP */
    int    ai_family;         /* e.g. PF_HIP */
    int    ai_socktype;       /* e.g. SOCK_STREAM */
    int    ai_protocol;       /* 0 or IPPROTO_HIP */
    size_t ai_addrlen;        /* size of *ai_addr  */
    struct sockaddr *ai_addr; /* sockaddr_hip */
    char   *ai_canonname;     /* canon. name of the host */
    struct addrinfo *ai_next; /* next endpoint */
};
```

Figure 3

The flag AI_HIP must be set in the ai_flags, or otherwise the resolver does not return sockaddr_hip data structures. The resolver returns EAI_BADFLAGS when AI_HIP is not supported. The simultaneous use of both AI_HIP and AI_PASSIVE flags equals to the use HIP_HIT_ANY macro as described in section Section 4.1. Similarly, the use of

AI_PASSIVE_PUB and AI_PASSIVE_ANON flag equals to the use of HIP_HIT_ANY_PUB and HIP_HIT_ANY_ANON.

The ai_family field is set to PF_HIP in the addrinfo structure when ai_addr points to a sockaddr_hip structure.

When ai_protocol field is set to zero, the resolver also returns locators in sockaddr_in and sockaddr_in6 structures in addition to sockaddr_hip structures. The resolver only returns sockaddr_hip structures when ai_protocol field is set to IPPROTO_HIP or a sockaddr_hip structure is given as the hint argument to the resolver.

A HIP-aware application creates the sockaddr_hip structures manually or obtains them from the resolver. The manual configuration is described in [I-D.ietf-shim6-multihome-shim-api]. This document defines resolver extensions for getaddrinfo resolver [RFC3493].

```
#include <netdb.h>

int getaddrinfo(const char *nodename,
                const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res)
void free_addrinfo(struct addrinfo *res)
```

Figure 4

As described in [RFC3493], the getaddrinfo function takes the nodename, servname, and hints as its input arguments. It places the result of the query into the res argument. The return value is zero on success, or a non-zero error value on error. The nodename argument specifies the host name to be resolved; a NULL argument denotes the local host. The servname parameter declares the port number to be set in the socket addresses in the res output argument. Both the nodename and servname cannot be NULL.

The input argument "hints" acts like a filter that defines the attributes required from the resolved endpoints. A NULL hints argument indicates that any kind of endpoints are acceptable.

The output argument "res" is dynamically allocated by the resolver. The application frees res argument with the free_addrinfo function. The res argument contains a linked list of the resolved endpoints. The linked list contains sockaddr_hip structures only when the input argument has the AI_HIP flag set. The resolver inserts HITs before any locators.

Resolver can return a HIT which maps to multiple locators. The

resolver may cache the locator mappings to the HIP module. The HIP module manages the multiple locators according to local policies of the host.

4.3. Manual Handling of Locators

The system resolver, or the HIP module, maps HITs to locators implicitly. However, some applications may want to specify initial locator mappings explicitly. In such a case, the application first creates a socket with PF_HIP as the domain argument. Second, the application binds the socket to a local or peer locator with the setsockopt function with either SHIM_LOC_LOCAL_PREF or SHIM_LOC_PEER_PREF as the socket option name as defined in [I-D.ietf-shim6-multihome-shim-api]. Third, the application creates a valid sockaddr_hip structure. Finally, the application associates the socket also with the sockaddr_hip structure by calling some socket-related function, such as connect or bind. The function returns EINVALIDLOCATOR when the HIT is not reachable at the specified locator.

It should be noticed that the application may just configure the HIT manually without setting the locator. In this scenario, the application relies on the system to map the HIT to an IP address. When the system fails to provide the mapping, it returns EADDRNOTAVAIL in the called sockets API function to the application and sets errno to indicate the error.

5. Summary of New Definitions

Table 3 summarizes the new macro and structures defined in this document.

Header	Definition
<sys/socket.h>	PF_HIP
<sys/socket.h>	AF_HIP
<netinet/in.h>	IPPROTO_HIP
<netinet/hip.h>	HIP_HIT_ANY
<netinet/hip.h>	HIP_HIT_ANY_PUB
<netinet/hip.h>	HIP_HIT_ANY_ANON
<netinet/hip.h>	HIP_IS_IPV6_ADDR_ORCHID
<netinet/hip.h>	HIP_IS_IPV6_ADDR_ANON_HIT
<netinet/hip.h>	HIP_FLAG_ONLY_ORCHID
<netdb.h>	AI_HIP
<netdb.h>	AI_PASSIVE_ANON
<netdb.h>	AI_PASSIVE_PUB
<netdb.h>	AI_HIP_NOLOCATORS
<netinet/hip.h>	hip_hit_t
<netinet/in.h>	sockaddr_hip

Table 3

6. IANA Considerations

No IANA considerations.

7. Security Considerations

No security considerations currently.

8. Acknowledgements

Jukka Ylitalo and Pekka Nikander have contributed many ideas, time and effort to the native HIP APIs. Thomas Henderson, Kristian Slavov, Julien Laganier, Jaakko Kangasharju, Mika Kousa, Jan Melen, Andrew McGregor, Sasu Tarkoma, Lars Eggert, Joe Touch, Antti Jaervinen, Anthony Joseph, Teemu Koponen, Jari Arkko, Ari Keraenen, Juha-Matti Tapio, Shinta Sugimoto, Philip Matthews, Jan Melen and Gonzalo Camarillo have also provided valuable ideas or feedback. Thanks for the APPS area folks, Stephane Bortzmeyer, Chris Newman, Tony Finch, "der Mouse" and Keith Moore for comments.

9. Normative References

[I-D.henderson-hip-applications]
Henderson, T. and P. Nikander, "Using HIP with Legacy Applications", draft-henderson-hip-applications-03 (work in progress), May 2006.

[I-D.ietf-btns-c-api]
Komu, M., "IPsec Application Programming Interfaces", draft-ietf-btns-c-api-01 (work in progress), July 2007.

[I-D.ietf-hip-base]
Moskowitz, R., Nikander, P., Jokela, P., and T. Henderson, "Host Identity Protocol", draft-ietf-hip-base-10 (work in progress), October 2007.

[I-D.ietf-shim6-multihome-shim-api]
Komu, M., "Socket Application Program Interface (API) for Multihoming Shim", draft-ietf-shim6-multihome-shim-api-03 (work in progress), July 2007.

[I-D.ietf-shim6-proto]
Bagnulo, M. and E. Nordmark, "Shim6: Level 3 Multihoming Shim Protocol for IPv6", draft-ietf-shim6-proto-09 (work in progress), November 2007.

[POSIX] Institute of Electrical and Electronics Engineers, "IEEE Std. 1003.1-2001 Standard for Information Technology - Portable Operating System Interface (POSIX)", Dec 2001.

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, February 2003.

[RFC4423] Moskowitz, R. and P. Nikander, "Host Identity Protocol (HIP) Architecture", RFC 4423, May 2006.

[RFC4843] Nikander, P., Laganier, J., and F. Dupont, "An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers (ORCHID)", RFC 4843, April 2007.

Author's Address

Miika Komu
Helsinki Institute for Information Technology
Metsaenneidonkuja 4
Helsinki
Finland

Phone: +358503841531
Fax: +35896949768
Email: miika@iki.fi
URI: http://www.iki.fi/miika/

Full Copyright Statement

THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).