Host Identity Protocol
Internet-Draft
Expires: August 30, 2005

M. Komu
Helsinki Institute for Information Technology

Mar 2005

Native Application Programming Interfaces for the Host Identity Protocol
draft-mkomu-hip-native-api-00.txt

Status of this Memo

This document is an Internet-Draft and is subject to all provisions of section 3 of RFC 3667. By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she become aware will be disclosed, in accordance with RFC 3668.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on August 30, 2005.

Copyright Notice

Copyright (C) The Internet Society (2005).

Abstract

This document proposes extensions to the current networking APIs. Using the extented APIs, HIP aware applications can gain a better control of the HIP layer and Host Identifiers. For example, the applications can query and set security and mobility related attributes, or specify their own Host Identifiers in a host.

Table of Contents

1. Introduction

Host Identity Protocol proposes a new cryptographic namespace and a new layer to the TCP/IP architecture. Applications can see these new changes in the networking stacks with varying degrees of visibility. [I-D.henderson-hip-applications] discusses the lowest levels of visibility in which applications are either completely or partially unaware of HIP. In this document, we discuss about the highest level of visibility. The applications are completely HIP aware and are

able to control the HIP layer and identifiers. The applications are allowed to query and set security related attributes and even specify their own Host Identifiers.

2. Design Architecture

In this section, the native HIP API design is described from an architectural point of view. We introduce the ED concept, which is a central idea in the API. We describe the layering and namespace models along with the socket bindings. We conclude the discussion with a description of the endpoint identifier resolution mechanism.

2.1 Endpoint Descriptor

The representation of endpoints is hidden from the applications. The ED is a ``handle'' to a HI. A given ED serves as a pointer to the corresponding HI entry in the HI database of the host. The ED is the AID [I-D.nordmark-multi6-noid] in the native HIP API model.

2.2 Layering Model

The application layer accesses the transport layer via the socket interface. The application layer uses the traditional TCP/IP IPv4 or IPv6 interface, or the new native HIP API interface provided by the socket layer. The layering model is illustrated in Figure 1. For simplicity, the IPsec layer has been excluded from the figure.

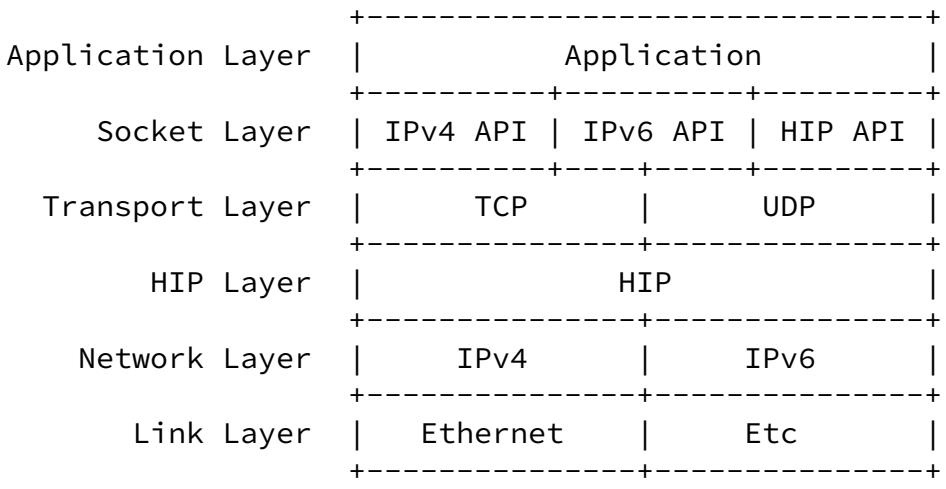


Figure 1

The HIP layer is as a shim/wedge layer between the transport and network layers. The datagrams delivered between the transport and network layers are intercepted in the HIP layer to see if the datagrams are HIP related and require HIP intervention.

2.3 Namespace Model

The used namespace model is shown in . The namespace identifiers are described in this section.

Layer	Identifier
User Interface	FQDN
Application Layer	ED, port and protocol
Transport Layer	HI, port
HIP Layer	HI
Network Layer	IP address

Table 1

People prefer human-readable names when referring to network entities. The most commonly used identifier in the UI is the FQDN, but there are also other ways to name network entities. The FQDN format is still the preferred UI level identifier in the context of the native HIP API.

In the current API, connection associations in the application layer are uniquely distinguished by the source IP address, destination IP address, source port, destination port, and protocol. HIP changes this model by using HIT in the place of IP addresses. The HIP model is further expanded in the native HIP API model by using ED instead of HITs. Now, the application layer uses source ED, destination ED, source port, destination port, and transport protocol type, to

distinguish between the different connection associations.

Basically, the difference between the application and transport layer identifiers is that the transport layer uses HIs instead of EDs. The TLI is named with source HI, destination HI, source port, and destination port at the transport layer.

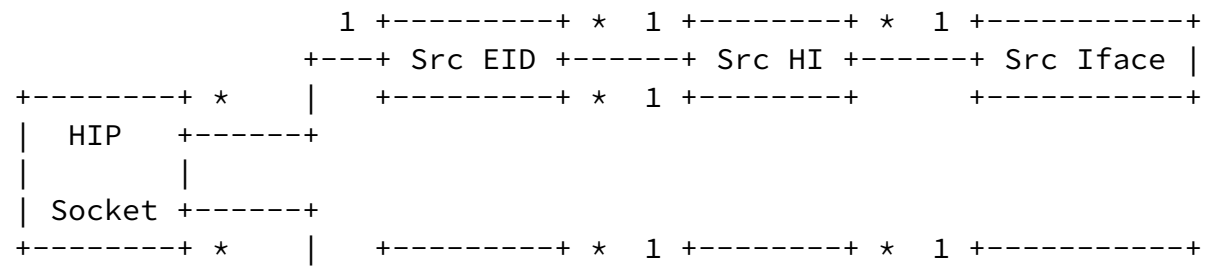
Correspondingly, the HIP layer uses HIs as identifiers. The HIP security associations are based on source HI and destination HI pairs.

The network layer uses IP addresses, i.e., locators, for routing purposes. The network layer interacts with the HIP layer to exchange information about changes in the local interfaces addresses and peer addresses.

2.4 Socket Bindings

A HIP socket is associated with one source and one destination ED,

along with their port numbers and protocol type. The relationship between a socket and ED is a many-to-one one. Multiple EDs can be associated with a single HI. Further, the source HI is associated with a set of network interfaces at the local host. The destination HI, in turn, is associated with a set of destination addresses of the peer. The socket bindings are visualized in Figure 2.



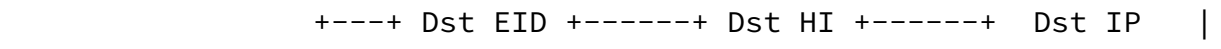


Figure 2

The relationship between a source ED and a source HI is always a many-to-one one. However, there are two refinements to the relationship. First, a listening socket is allowed to accept connections from all local HIs of the host. Second, the opportunistic mode allows the base exchange to be initiated to an unknown destination HI. In a way, the relationship between the local ED and local HI is a many-to-undefined relationship for a moment in both of the cases, but once the connection is established, the ED will be permanently associated with a certain HI.

The ED concept can only be used in HIP protocol family sockets. Other types of sockets are left intact to avoid breaking the backwards compatibility.

The DNS based endpoint discovery mechanism is illustrated in . The application calls the resolver (step a.) to resolve an FQDN (step b.). The DNS server responds with a HI and a set of IP addresses (step c.). The resolver does not directly pass the HI and the locators to the application, but sends them to the HIP module (step d.). Finally, the resolver receives an ED from the HIP module (step e.) and passes the ED to the application (step f.).

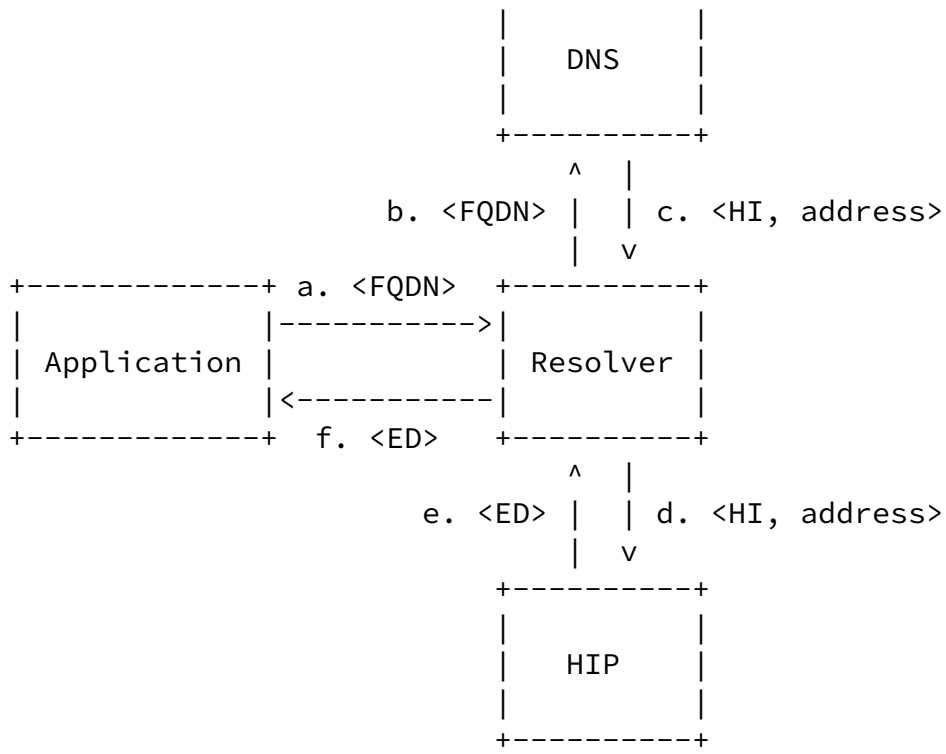


Figure 3

The application can also receive multiple EDs from the resolver if the FQDN is associated with multiple HIs. The endpoint discovery mechanism is still almost the same. The difference is that the DNS returns a set of HIs (along with their locators) to the resolver. The resolver sends all of them to the HIP module and receives a set of EDs in return, each ED corresponding to a single HI. Finally, the EDs are sent to the application.

3. IANA Considerations

To be done.

4. Security Considerations

To be done.

5. Acknowledgements

Jukka Ylitalo and Pekka Nikander have contributed many ideas, time and effort to the native HIP API. Thomas Henderson, Kristian Slavov, Julien Laganier, Jaakko Kangasharju, Mika Kousa, Jan Melen, Andrew McGregor, Sasu Tarkoma, Lars Eggert, Joe Touch, Antti J?rvinen and Anthony Joseph have also provided valuable ideas and feedback.

6. References

6.1 Normative References

[I-D.ietf-hip-base]
Moskowitz, R., "Host Identity Protocol", draft-ietf-hip-base-02 (work in progress), February 2005.

[I-D.ietf-hip-mm]
Nikander, P., "End-Host Mobility and Multi-Homing with Host Identity Protocol", draft-ietf-hip-mm-01 (work in progress), February 2005.

[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.

6.2 Informative References

[I-D.henderson-hip-applications]
Henderson, T., "Using HIP with Legacy Applications", draft-henderson-hip-applications-00 (work in progress), February 2005.

[I-D.nordmark-multi6-noid]
Nordmark, E., "Multihoming without IP Identifiers", draft-nordmark-multi6-noid-02 (work in progress), July 2004.

Author's Address

Miika Komu
Helsinki Institute for Information Technology
Tammasaarenkatu 3
Helsinki
Finland

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

Appendix A. Interface Syntax and Description

In this section, we describe the native HIP API using the syntax of the C programming language and present only the ``external'' interfaces and data structures that are visible to the applications. We limit the description to those interfaces and data structures that are either modified or completely new, because the native HIP API is otherwise identical to the sockets API [POSIX].

A.1 Data Structures

We introduce a new protocol family, PF_HIP, for the sockets API. The AF_HIP constant is an alias for it. The use of the PF_HIP constant is mandatory with the socket function if the native HIP API is to be used in the application. The PF_HIP constant is given as the first argument (domain) to the socket function.

The ED abstraction is realized in the sockaddr_ed structure, which is shown in figure Figure 4. The family of the socket, ed_family, is set to PF_HIP. The port number ed_port is two octets and the ED value ed_val is four octets. The ED value is just an opaque number to the application. The application should not try to associate it directly to a HI or even compare it to other ED values, because there are separate functions for those purposes. The ED family is stored in host byte order. The port and the ED value are stored in network byte order.

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

Figure 4

The ed_val field is usually set by special native HIP API functions, which are described in the following section. However, three special macros can be used to directly set a value into the ed_val field. The macros are HIP_HI_ANY, HIP_HI_ANY_PUB and HIP_HI_ANY_ANON. They denote an ED value associated with a wildcard HI of any, public, or anonymous type. This is useful to a ``server'' application that is willing to accept connections to all of the HIs of the host. The macros correspond to the sockets API macros INADDR_ANY and IN6ADDR_ANY_INIT, but they are applicable on the HIP layer. It should be noted that only one process at a time can bind with the HIP_HI_*ANY macro on a certain port to avoid ambiguous bindings.

The native HIP API has a new resolver function which is used for

querying both endpoint identifiers and locators. The resolver introduces a new data structure, which is used both as the input and output argument for the resolver. The new structure, endpointinfo, is shown in Figure 5.

```
 struct endpointinfo {
        int ei_flags;                 /* flags, e.g. EI_FALLBACK */
        int ei_family;                /* e.g. PF_HIP */
        int ei_socktype;              /* e.g. SOCK_STREAM */
        int ei_protocol;              /* usually just zero */
        size_t ei_endpoint_len;       /* length of the endpoint */
        struct sockaddr *ei_endpoint; /* endpoint socket address */
        char *ei_canonname;           /* canonical name of the host */
        struct endpointinfo *ei_next; /* next endpoint */
};
```

Figure 5

The members of the endpointinfo structure are similar to addrinfo structure, but the member names have a different prefix. The socket address structure used for sockets API calls has been renamed to ei_endpoint to emphasize the difference with the getaddrinfo

resolver. The family, ei_family, is set to PF_HIP when the socket address structure contains an ED that refers to a HI.

The flags in the endpointinfo structure control the behavior of the resolver and describe the attributes of the endpoints and locators. The EI_ANON flag forces the resolver to query only for local anonymous identifiers. The default action is first to resolve the public endpoints and then the anonymous endpoints.

Some applications may prefer configuring the locators manually and can set the EI_NOLOCATORS to prohibit the resolver from resolving any locators. If the application wants to configure locators manually, the EI_NOLOCATORS flag forces the resolver to discard the resolving of locators. The EI_FALLBACK flag suggests the resolver to return locators if no HIs are found. The ei_endpoint members in the resolver output are then filled with IPv4 or IPv6 addresses and the application can resort to plain TCP/IP connections using the IP addresses returned. The fallback flag must be explicitly enabled in the flags, because the resolver returns only HIs by default. The EI_HI_ANY, EI_HI_ANY_PUB and EI_HI_ANY_ANON flags cause the resolver to output only a single socket address containing an ED that would be received using the corresponding HIP_HI_*ANY macro.

Application specified endpoint identifiers are essentially private keys. To support application specified identifiers in the API, we need new data structures for storing the private keys. The private

keys need an uniform format so that they can be easily used in API calls. The keys are stored in the endpoint structures shown in figure Figure 6.

```
struct endpoint {
        se_length_t   length;
        se_family_t   family;
};
struct endpoint_hip {
```

```
                se_length_t length;
                se_family_t family;   /* EF_HI */
                se_hip_flags_t flags;
                union {
                        struct hip_host_id host_id;
                hit_t hit;
                } id;
        };
```

Figure 6

The structure endpoint represents a generic endpoint and the endpoint_hip is the HIP specific endpoint. The HIP endpoint is public by default unless HIP_ENDPOINT_FLAG_ANON flag is set in the structure to anonymize the endpoint. The id union contains the HI in the host_id member in the format specified in the HIP draft [I-D.ietf-hip-base]. The draft does not specify the format for the private key, so private key material is just appended to the host_id and the length is adjusted accordingly. The flag HIP_ENDPOINT_FLAG_PRIVATE is also set. The hit member of the union is used only when the HIP_ENDPOINT_FLAG_HIT flag is set.

An optional extension to the getaddrinfo interface is introduced too. A new flag, AI_HIP_RVS, is used both in the input and output of the resolver. By default, the getaddrinfo resolver does not return IP addresses belonging to a HIP rendezvous server. The resolver returns rendezvous server addresses only when the AI_HIP_RVS flag is set in the resolver hints. This way, legacy applications can never receive any addresses belonging to a rendezvous server. The flag is also set in the getaddrinfo resolver output to denote that the resolved address belongs to a HIP rendezvous server.

A.2 Functions

The new functions introduced to the sockets API are described in this section.

A.2.1 Resolver Interface

The native HIP API does not introduce changes to the interface syntax of the fundamental sockets API functions bind, connect, send, sendto, sendmsg, recv, recvfrom, and recvmsg. The application usually calls the functions with sockaddr_ed structures instead of sockaddr_in or sockaddr_in6 structures. The source of the sockaddr_ed structures in the native HIP API is the resolver function getendpointinfo which is shown in Figure 7.

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

Figure 7

The getendpointinfo 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 sets the port number to be set in the socket addresses in the res output argument. Both the nodename and servname cannot be NULL.

The output argument res is dynamically allocated by the resolver. The application must free it with the free_endpointinfo function. It contains a linked list of the resolved endpoints. The input argument hints acts like a filter that defines the attributes required from the resolved endpoints. For example, setting the flag HIP_ENDPOINT_FLAG_ANON in the hints forces the resolver to return only anonymous endpoints in the output argument res. If the hints argument is zero, any kind of endpoints are acceptable.

A.2.2 Application Specified Identities

Application specified local and peer endpoints can be retrieved from

files using the function shown in Figure 8. The function hip_endpoint_load_pem is used for retrieving a private or public key from a given file filename. The file must be in PEM encoded format. The result is allocated dynamically and stored into the endpoint argument. The return value of the function is zero on success, or a non-zero error value on failure. The result is deallocated with the free system call.

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

Figure 8

The endpoint structure cannot be used directly in the sockets API function calls. The application must convert the endpoint into an ED first. Local endpoints are converted with the getlocaled function and peer endpoints with getpeered function. The functions are illustrated in Figure 9. Both of functions are used in a similar way.

```
struct sockaddr_ed *getlocaled(const struct endpoint *endpoint,
                               const char *servname,
                               const struct addrinfo *addrs,
                               const struct if_nameindex *ifaces,
                               int flags)
struct sockaddr_ed *getpeered(const struct endpoint *endpoint,
                              const char *servname,
                              const struct addrinfo *addrs,
                              int flags)
```

Figure 9

The result of the conversion, an ED socket address, is returned by the functions. A failure in the conversion causes a NULL return value to be returned and the errno to be set accordingly. The caller of the functions is responsible of freeing the returned socket address structure.

The endpoint argument is retrieved e.g. with the hip_endpoint_load_pem function. If the endpoint is NULL, an arbitrary HI of the host is selected and associated with the ED value of the third argument.

The servname argument is the service string. The function converts it to a numeric port number and fills the port number into the returned ED socket structure for the convenience of the application.

The addrs argument defines the initial IP addresses of the local host or peer host. The argument is a pointer to a linked list of addrinfo structures containing the initial addresses of the peer. The list pointer can be obtained with a getaddrinfo [RFC3493] function call. A NULL pointer indicates that the application trusts the host to already know the locators of the peer. We recommend that a NULL pointer is not given to the getpeered function to ensure reachability with the peer.

The getlocaled function accepts also a list of network interface indexes in the ifaces argument. The list can be obtained with the if_nameindex [RFC3493] function call. A NULL list pointer indicates all the interfaces of the local host. Both the IP addresses and interfaces can be combined to select a specific address from a specific interface.

The last argument is the flags. The following flags are valid only for the getlocaled function:

- HIP_ED_*ANY correspond to the use of the HIP_HI_*ANY macros.
- Flags HIP_HI_REUSE_UID, HIP_HI_REUSE_GID and HIP_HI_REUSE_ANY

allow the HI binding to be reused for processes with the same UID, GID or any UID as the calling process.

- Flags HIP_ED_IP and HIP_ED_IPV6 are used for limiting the address family scope of the interfaces.

It should noticed that the HIP_HI_ANY, HIP_HI_ANY_PUB and HIP_HI_ANY_ANON macros can be defined as calls to the getlocaled call with a NULL endpoint, NULL interface, NULL address argument and the flag corresponding to the macro name set.

A.2.3 Querying Endpoint Related Information

The getlocaled and getpeered functions have also their reverse counterparts. Given an ED, the getlocaledinfo and getpeeredinfo functions search for the HI and the current set of locators associated with the ED. The first argument is the ED to be searched for. The functions write the results of the search, the HIs and locators, to the rest of the function arguments. The function interfaces are depicted in figure Figure 10. The caller of the functions is responsible for freeing the memory reserved for the search results.

```
int getlocaledinfo(const struct sockaddr_ed *my_ed,
                   struct endpoint **endpoint,
                   struct addrinfo **addrs,
                  struct if_nameindex **ifaces)
int getpeeredinfo(const struct sockaddr_ed *peer_ed,
                  struct endpoint **endpoint,
                  struct addrinfo **addrs)
```

Figure 10

The getlocaledinfo and getpeeredinfo functions are especially useful for an advanced application that receives multiple EDs from the resolver. The advanced application can query the properties of the EDs using getlocaledinfo and getpeeredinfo functions and select the ED that matches the desired properties.

A.2.4 Socket Options

As usually, getting and setting of HIP socket options is done using getsockopt and setsockopt functions. To set HIP layer specific socket options, the first argument must be a socket descriptor that was instantiated with PF_HIP as the domain, and the second argument must be specified as IPPROTO_HIP

Some HIP socket option names are listed in Table 2. The length of the option must be natural word size of the underlying processor, typically 32 or 64 bits. The purpose of the option value must be interpreted in context of the protocol specifications [I-D.ietf-hip-base][I-D.ietf-hip-mm].

The socket options must be set before the hosts have established HIP SA. The implementation may refuse to set the socket options if there is already an existing SA associated with the given socket.

Socket Options	Purpose
SO_HIP_CHALLENGE_SIZE	Puzzle challenge size
SO_HIP_HIP_TRANSFORMS	Integer array of the preferred HIP transforms
SO_HIP_ESP_TRANSFORMS	Integer array of the preferred ESP transforms
SO_HIP_DH_GROUP_IDS	Integer array of the preferred Diffie-Hellman group IDs
SO_HIP_SA_LIFETIME	Socket association lifetime in seconds
SO_HIP_RETRANS_INIT_TIMEOUT	HIP initial retransmission timeout
SO_HIP_RETRANS_INTERVAL	HIP retransmission interval in seconds
SO_HIP_RETRANS_ATTEMPTS	Number of retransmission

		attempts
	SO_HIP_AF_FAMILY	The preferred IP address family. The default family is AF_ANY.

	SO_HIP_PIGGYPACK	If set to one, HIP piggy-packing is preferred. Zero if piggy-packing must not be used.
	SO_HIP_OPPORTUNISTIC	Try HIP in opportunistic mode if only the locators of the peer are known.
	SO_HIP_OPP_FALLBACK	The same as above, but fall back to plain TCP/IP if base exchange failed
	SO_HIP_BEX_FALLBACK	Try normal base exchange, but fall back to plain TCP/IP if the base exchange fails.

Table 2

Intellectual Property Statement

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.

Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND 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.

Copyright Statement

Copyright (C) The Internet Society (2005). This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.