Dual Stack Hosts Using "Bump-in-the-API" (BIA)
draft-huang-behave-rfc3338bis-00

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79. This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

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.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.
Abstract

This document describes the "Bump-In-the-API" (BIA) host based protocol translation mechanism that allows applications supporting only one IP address family to communicate with peers that are reachable or supporting only the other address family.

This specification addresses scenarios where a host is provided dual stack, IPv6 only or IPv4 only network connectivity. In the dual stack network case, single address family applications in the host will communicate directly with other hosts using the same address family. In the case of IPv6 only network or IPv6 only destination, IPv4-originated communications have to be translated into IPv6. IPv6 communications may have to be translated similarly to IPv4. Technically, the BIS-enabled host resolves both IPv4 and IPv6 addresses of the destination and behaves according to received responses.

Acknowledgement of previous work

This document is an update to and directly derivative from Seungyun Lee, Myung-Ki Shin, Yong-Jin Kim, Alain Durand, and Erik Nordmark's [RFC3338], which similarly provides a dual stack host means to communicate with other IPv6 host using existing IPv4 applications. The original document was a product of the NGTRANS working group.

The changes in this document reflect three components:

1. Supporting IPv6 only network connections
2. Supporting IPv4 only network connections
3. Supporting Well Known Prefix
4. Removing Applicability and Disclaimer

The goal of this mechanism is the same as that of the Bump-in-the-stack mechanism, but this mechanism provides the translation method between the IPv4 APIs and IPv6 APIs. Thus, the goal is simply achieved without IP header translation.
Table of Contents

1. Introduction .......................... 3
2. Dual Stack Host Architecture Using BIA .......................... 5
   2.1 Function Mapper .......................... 6
   2.2 Name Resolver .......................... 6
   2.3 Address Mapper .......................... 7
3. Behavior Examples -- dual stack network and IPv6 only peer ... 8
   3.1 Originator Behavior .......................... 8
   3.2 Recipient Behavior .......................... 10
4. Behavior Examples -- IPv6 only network and dual-stack peer .. 11
   4.1 Originator Behavior .......................... 12
   4.2 Recipient Behavior .......................... 14
5. Behavior Examples -- IPv4 only network and IPv4 only peer .. 14
   5.1 Originator Behavior .......................... 15
   5.2 Recipient Behavior .......................... 17
6. Considerations .......................... 19
   6.1 Socket API Conversion .......................... 19
   6.2 ICMP Message Handling .......................... 19
   6.3 IPv4 or IPv6 Address Pool and Mapping Table .......................... 19
   6.4 Internally Assigned IPv4 or IPv6 Addresses .......................... 19
   6.5 Mismatch between DNS result(AAAA) and Peer Application ... 20
      Version(v4) .......................... 20
   6.6 Implementation Issues .......................... 20
7. Limitations .......................... 22
8. Security Considerations .......................... 22
9. Acknowledgments .......................... 22
10. References .......................... 22
Appendix A : API list intercepted by BIA .......................... 25
Authors’ Addresses .......................... 27

1. Introduction

RFC3338 [RFC3338] stated that there are few applications for IPv6
[IPV6] as compared with IPv4 [IPV4] in which a great number of
applications are available. In order to advance the transition
smoothly, it is highly desirable to make the availability of IPv6
applications increase to the same level as IPv4. Unfortunately,
however, this is expected to take a long time. Meanwhile, there are
scenarios where a dual stack host is connected to IPv6-only network
but it is running IPv4-only applications, or a host is running IPv6-
only applications while connected to IPv4-only network.

RFC3338 proposed a mechanism of dual stack hosts using the technique
called "Bump-in-the-API" [BIA] in the IP security area. The
technique inserts an API translator between the socket API module and
the TCP/IP module in the dual stack hosts, so that it translates the
IPv4 socket API function into IPv6 socket API function and vice versa.

RFC2767 [BIS] specifies a host translation mechanism using a technique called "Bump-in-the-Stack". It translates IPv4 into IPv6, and vice versa using the IP conversion mechanism defined in [SIIT]. BIS allows hosts to communicate with other IPv6 hosts using existing IPv4 applications. However, this approach is to use an API translator which is inserted between the TCP/IP module and network card driver, so that it has the same limitations as the [SIIT] based IP header translation methods. In addition, its implementation is dependent upon the network interface driver.

When IPv4 or IPv6 applications on the dual stack communicate with other IPv6 or IPv4 hosts, the API translator detects the socket API functions from IPv4 or IPv6 applications and invokes the IPv6 or IPv4 socket API functions to communicate with the IPv6 or IPv4 hosts, and vice versa. In order to support communication between IPv4 or IPv6 applications and the target IPv6 or IPv4 hosts, pooled IPv4 or IPv6 addresses will be assigned through the name resolver in the API translator. But the those IPv4 or IPv6 addresses never flow out from them.

The network scenario specified in RFC3338 is a dual stack network, where IPv4 communication can be transported independently of IPv6. However, if the network provides only IPv6 transport, applications’s IPv4 packets have to be translated into IPv6. The opposite happens when the network is IPv4-only and application is IPv6-only capable.

This specification assumes that host knows it is connected with a dual stack network, IPv6-only network or IPv4-only network. The host learns that from layer 2 or from results of layer 3 IP address configuration mechanisms.

If the network which host is connecting with is IPv4 only network, then host’s IPv4 application will behave regularly, and it’s IPv6 application’s packets have to be translated into IPv4 packets.

If the network which host is connecting with is IPv6 only network, then host’s IPv6 application will behave regularly, and it’s IPv4 application’s packets have to be translated into IPv6 in order to communicate with IPv6 applications.

If the network which host is connecting with is dual stack network, then host will behave as what RFC3338 originally described.

The scenario where destination peer is not reachable with the address family a host is provisioned with is not covered by this document, as
that requires network based protocol translation solution. However, the BIA technology can complement network based protocol translation such as [NAT64] and [PNAT].

Moreover, since the translation is automatically carried out with help of DNS protocol, most applications do not need to know whether target hosts are IPv6 or IPv4 ones. That is, this allows hosts to communicate with other IPv6 hosts using existing IPv4 applications and other IPv4 hosts using existing IPv6 applications; thus it seems as if peers are always dual stack hosts with applications for both IPv4 and IPv6.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119].

This document uses terms defined in [IPv6], [TRANS-MECH], [BIS] and [BIA].

2. Dual Stack Host Architecture Using BIA

Figure 1 shows the architecture of the host in which BIA is installed.

```
+----------------------------------------------+
  +-------------------+  +-------------------+ |
  |                  |  |                   | |
  | IPv4 applications |  | IPv6 applications |
  |                  |  |                   | |
  +-------------------+  +-------------------+ |
  |                  |  |                   | |
  | Socket API (IPv4, IPv6) |               |
  |                  |  |                   | |
  +-------------------+  +-------------------+ |
  |                  |  |                   | |
  | [ API translator]------------------------+ |
  |                  |  |                   | |
  | +-----------+ +---------+ +------------+ | |
  | | Name      | | Address | | Function  | |
  | | Resolver  | | Mapper  | | Mapper  | |
  | +-----------+ +---------+ +------------+ | |
  +------------------------------------------+ |
  | +--------------------+ +-------------------+ |
  | |                    | |                   | |
  | | TCP(UDP)/IPv4      | |   TCP(UDP)/IPv6   |
  | |                    | |                   | |
  +--------------------+ +-------------------+ |
  +----------------------------------------------+
```

Figure 1 Architecture of the dual stack host using BIA
Dual stack hosts defined in RFC2893 [TRANS-MECH] need applications, TCP/IP modules and addresses for both IPv4 and IPv6. The proposed hosts in this document have an API translator to communicate with other IPv6 hosts using existing IPv4 applications, or communicate with other IPv4 hosts using existing IPv4 applications. The API translator consists of 3 modules, a name resolver, an address mapper and a function mapper.

2.1 Function Mapper

It translates an IPv4 socket API function into an IPv6 socket API function, and vice versa.

When detecting the IPv4 socket API functions from IPv4 applications, it intercepts the function call and invokes new IPv6 socket API functions which correspond to the IPv4 socket API functions. Those IPv6 API functions are used to communicate with the target IPv6 hosts. When detecting the IPv6 socket API functions from the data received from the IPv6 hosts, it works symmetrically in relation to the previous case.

When detecting the IPv6 socket API functions from IPv6 applications, it intercepts the function call and invokes new IPv4 socket API functions which correspond to the IPv6 socket API functions. Those IPv4 API functions are used to communicate with the target IPv4 hosts. When detecting the IPv4 socket API functions from the data received from the IPv4 hosts, it works symmetrically in relation to the previous case.

2.2 Name Resolver

It returns a proper answer in response to the IPv4 or IPv6 application’s request.

When an IPv4 application tries to resolve names via the resolver library (e.g. gethostbyname()), BIA intercept the function call and instead call the IPv6 equivalent functions (e.g. getnameinfo()) that will resolve both A and AAAA records.

If the AAAA record is available, it requests the address mapper to assign an IPv4 address corresponding to the IPv6 address, then creates the A record for the assigned IPv4 address, and returns the A record to the application.

When an IPv6 application tries to resolve names via the resolver library (e.g. getnameinfo()), BIA intercept the function call and instead call the IPv4 equivalent functions (e.g. gethostbyname()) that will resolve both A and AAAA records.
If the A record is available, it requests the address mapper to assign an IPv6 address corresponding to the IPv4 address, then creates the AAAA record for the assigned IPv6 address, and returns the AAAA record to the application.

### 2.3 Address Mapper

It internally maintains a table of the pairs of an IPv4 address and an IPv6 address. The IPv4 addresses are assigned from an IPv4 address pool. It uses the unassigned IPv4 addresses (e.g., 0.0.0.1 ~ 0.0.0.255).

When the name resolver or the function mapper requests it to assign an IPv4 address corresponding to an IPv6 address, it selects and returns an IPv4 address out of the pool, and registers a new entry into the table dynamically. The registration occurs in the following 2 cases:

1. When the name resolver gets only an ‘AAAA’ record for the target host name and there is not a mapping entry for the IPv6 address.

2. When the function mapper gets a socket API function call from the data received and there is not a mapping entry for the IPv6 source address.

When the name resolver or the function mapper requests it to assign an IPv6 address corresponding to an IPv4 address, it selects and returns an IPv6 address out of the pool, and registers a new entry into the table dynamically. The registration occurs in the following 2 cases:

1. When the name resolver gets only an ‘A’ record for the target host name and there is not a mapping entry for the IPv4 address.

2. When the function mapper gets a socket API function call from the data received and there is not a mapping entry for the IPv4 source address.

NOTE: This is the same as that of the Address Mapper in [BIS].

[BIS]
3. Behavior Examples -- dual stack network and IPv6 only peer

This section describes behaviors of the proposed dual stack host called "dual stack", which communicates with an IPv6 host called "host6" using an IPv4 application on dual stack network.

In this section, the meanings of arrows are as follows:

--- > A DNS message for name resolving created by the applications and the name resolver in the API translator.

+++ > An IPv4 or IPv6 address request to and reply from the address mapper for the name resolver and the function mapper.

=== > Data flow by socket API functions created by the applications and the function mapper in the API translator.

3.1 Originator Behavior

This sub-section describes the behavior when the "dual stack" sends data to "host6".

When an IPv4 application sends a DNS query to its name server, the name resolver intercepts the query and then creates a new query to resolve both A and AAAA records. When only the AAAA record is resolved, the name resolver requests the address mapper to assign an IPv4 address corresponding to the IPv6 address.

The name resolver creates an A record for the assigned IPv4 address and returns it to the IPv4 applications.

In order for the IPv4 application to send IPv4 packets to host6, it calls the IPv4 socket API function.

The function mapper detects the socket API function from the application. If the result is from IPv6 applications, it skips the translation. In the case of IPv4 applications, it requires an IPv6 address to invoke the IPv6 socket API function, thus the function mapper requests an IPv6 address to the address mapper. The address mapper selects an IPv4 address from the table and returns the destination IPv6 address. Using this IPv6 address, the function mapper invokes an IPv6 socket API function corresponding to the IPv4 socket API function.

When the function mapper receives an IPv6 function call, it requests the IPv4 address to the address mapper in order to translate the IPv6 socket API function into an IPv4 socket API function. Then, the function mapper invokes the socket API function for the IPv4 applications.
Figure 2 illustrates the behavior described above:

<table>
<thead>
<tr>
<th>&quot;dual stack&quot;</th>
<th>&quot;host6&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4 Socket</td>
<td>[ API Translator ]</td>
</tr>
<tr>
<td>appli- API</td>
<td>Name Address Function</td>
</tr>
<tr>
<td>API Resolver Mapper Mapper</td>
<td></td>
</tr>
</tbody>
</table>

<<Resolve an IPv4 address for "host6".>>

| -------- | ------ |  Query of 'A' records for host6. |
| -------- | ------ |  Query of 'A' records and 'AAAA' for host6 |
| | |  Reply with the 'AAAA' record. |

<<The 'AAAA' record is resolved.>>

| ++++++++ | Request one IPv4 address corresponding to the IPv6 address. |
| |  Assign one IPv4 address. >> |
| | |  Reply with the IPv4 address. |

<<Create 'A' record for the IPv4 address.>>

| -------- | ------ |  Reply with the 'A' record. |

Figure 2 Behavior of the originator (1/2)
### 3.2 Recipient Behavior

This subsection describes the recipient behavior of "dual stack". The communication is triggered by "host6".

"host6" resolves the address of "dual stack" with ‘AAAA’ records through its name server, and then sends an IPv6 packet to the "dual stack".

The IPv6 packet reaches the "dual stack" and the function mapper detects it.
The function mapper requests the IPv4 address to the address mapper in order to invoke the IPv4 socket API function to communicate with the IPv4 application. Then the function mapper invokes the corresponding IPv4 socket API function for the IPv4 applications corresponding to the IPv6 functions.

Figure 3 illustrates the behavior described above:

```
<table>
<thead>
<tr>
<th>&quot;dual stack&quot;</th>
<th>&quot;host6&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4 Socket</td>
<td>[ API Translator ]</td>
</tr>
<tr>
<td>appli- API</td>
<td>Name</td>
</tr>
<tr>
<td>cation</td>
<td>Resolver</td>
</tr>
</tbody>
</table>
<<Receive data from "host6".>>

An IPv6 Socket function call.  <<==-----------------==

<+++++++ Request IPv4 addresses corresponding to the IPv6 addresses.

+++++++> Reply with the IPv4 addresses.

<<Translate IPv6 into IPv4.>>

<==-----------------== An IPv4 function call

<<Reply an IPv4 data to "host6".>>

----------- An IPv4 function call

<<<<<< Translate IPv4 into IPv6.>>>

<<<<<< Request IPv6 addresses corresponding to the IPv4 addresses.

<<<<<<> Reply with the IPv6 addresses.

An IPv6 Socket function call. <==-----------------==

Figure 3 Behavior of Receiving data from IPv6 host
```

4. Behavior Examples -- IPv6 only network and dual-stack peer

This section describes behaviors of the proposed dual stack host called "dual stack", which communicates with a dual-stack peer
called "host46" using an only IPv4 application while provisioned only with IPv6 network connectivity.

4.1 Originator Behavior

This subsection describes the originator behavior of "dual stack."

The communication is triggered by "dual stack."

The application sends a query to its name server to resolve 'A' records for "host46."

The resolver snoops the query, then creates another query for 'AAAA' to resolve both 'A' and 'AAAA' records for the host name, and sends it to the server. In this case, both the 'A' and 'AAAA' records are resolved, so the resolver does not need to request the mapper to allocate any IPv4 addresses from its pool, but only to store mapping between received destination's IPv4 and IPv6 addresses.

In this case of communication with a dual-stack host, the 'A' record is also resolved and the resolver can return it to the application as is.

In order for the IPv4 application to send IPv4 packets to host6, it calls the IPv4 socket API function.

The function mapper detects the socket API function from the application. If the result is from IPv6 applications, it skips the translation. In the case of IPv4 applications, it requires an IPv6 address to invoke the IPv6 socket API function, thus the function mapper requests an IPv6 address to the address mapper. The address mapper selects an IPv4 address from the table and returns the destination IPv6 address. Using this IPv6 address, the function mapper invokes an IPv6 socket API function corresponding to the IPv4 socket API function.

When the function mapper receives an IPv6 function call, it requests the IPv4 address to the address mapper in order to translate the IPv6 socket function into an IPv4 socket API function. Then, the function mapper invokes the socket API function for the IPv4 applications.

Figure 4 illustrates the behavior described above:
|--------|------->| Query of ‘A’ records for host6. |
|--------|------->| Query of ‘A’ records and ‘AAAA’ for host6 |
|<-------|--------| Reply with the ‘AAAA’ record. |
|<-------|--------| Reply with the ‘AAAA’ record. |
|<-------|--------| Reply with the ‘A’ record. |

<<The ‘AAAA’ record is resolved.>>

+++++++> Request one IPv4 address corresponding to the IPv6 address.

<<Assign one IPv4 address.>>

+++++++> Reply with the IPv4 address.

<<Create ‘A’ record for the IPv4 address.>>

Figure 4 Behavior of the originator (1/2)
4.2 Recipient Behavior

The recipient behaviour is exactly the same as in 3.2.

5. Behavior Examples -- IPv4 only network and IPv4 only peer

This section describes action of the proposed dual stack host called "dual stack," which communicates with an IPv4 peer called "host4" using an IPv6 application.
5.1 Originator Behavior

This sub-section describes the behavior when the "dual stack" sends data to "host4".

When an IPv6 application sends a DNS query to its name server, the name resolver intercepts the query and then creates a new query to resolve both A and AAAA records. When only the A record is resolved, the name resolver requests the address mapper to assign an IPv6 address corresponding to the IPv4 address.

The name resolver creates an AAAA record for the assigned IPv6 address and returns it to the IPv6 applications.

In order for the IPv6 application to send IPv6 packets to host4, it calls the IPv6 socket API function.

The function mapper detects the socket API function from the application. If the result is from IPv4 applications, it skips the translation. In the case of IPv6 applications, it requires an IPv4 address to invoke the IPv4 socket API function, thus the function mapper requests an IPv4 address to the address mapper. The address mapper selects an IPv6 address from the table and returns the destination IPv4 address. Using this IPv4 address, the function mapper invokes an IPv4 socket API function corresponding to the IPv6 socket API function.

When the function mapper receives an IPv4 function call, it requests the IPv6 address to the address mapper in order to translate the IPv4 socket API function into an IPv6 socket API function. Then, the function mapper invokes the socket API function for the IPv6 applications.

Figure 6 illustrates the behavior described above:

```
"dual stack"                   "host4"
IPv6 Socket                  [ API Translator ] TCP(UDP)/IP Name
application                  Name Address Function (v6/v4) Server

<p>| | | | |
|                  |             |             |             |</p>
<table>
<thead>
<tr>
<th>&lt;Resolve an IPv6 address for &quot;host4&quot;.&gt;&gt;</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Query of ‘AAAA’ records for host4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Query of ‘A’ records and ‘AAAA’ for host4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;-----------------------------</td>
<td>-------------</td>
<td>-------------</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 6 Behavior of the originator (1/2)
5.2 Recipient Behavior

This subsection describes the recipient behavior of "dual stack". The communication is triggered by "host4".

"host4" resolves the address of "dual stack" with 'A' records through its name server, and then sends an IPv4 packet to the "dual stack".

The IPv4 packet reaches the "dual stack" and the function mapper detects it.
The function mapper requests the IPv6 address to the address mapper in order to invoke the IPv6 socket API function to communicate with the IPv6 application. Then the function mapper invokes the corresponding IPv6 socket API function for the IPv6 applications corresponding to the IPv4 functions.

Figure 7 illustrates the behavior described above:

```
"dual stack"                          "host4"
IPv6  Socket |  [ API Translator ]  | TCP(UDP)/IP
appli- API | Name  Address  Function | (v6/v4)
cation      | Resolver  Mapper  Mapper |

<<Receive data from "host4".>>

| An IPv4 Socket function call. <<==|==|==|==|==|==|
|<+++++++ Request IPv6 addresses corresponding to the IPv4 addresses. |
|<+++++++> Reply with the IPv6 addresses. |
|<Translate IPv4 into IPv6.>> |

<-------|-------|-------|-------|-------|-------|-------|-------|-------|
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
</tbody>
</table>

<<Reply an IPv6 data to "host4".>>

| An IPv6 function call |<==|==|==|==|==|==|
|<-------------|-------------|-------------|-------------|-------------|-------------|-------------|
|<Translate IPv6 into IPv4.>> |
|<+++++++ Request IPv4 addresses corresponding to the IPv6 addresses. |
|<+++++++> Reply with the IPv4 addresses. |
| An IPv4 Socket function call. <<==|==|==|==|==|==|
```

Figure 7 Behavior of Receiving data from IPv4 host
6. Considerations

6.1 Socket API Conversion

IPv4 socket API functions are translated into semantically the same IPv6 socket API functions and vice versa. See Appendix A for the API list intercepted by BIA. IP addresses embedded in application layer protocols (e.g., FTP) can be translated in API functions. Its implementation depends on operating systems.

NOTE: Basically, IPv4 socket API functions are not fully compatible with IPv6 since the IPv6 has new advanced features.

6.2 ICMP Message Handling

When an application needs ICMP messages values (e.g., Type, Code, etc.) sent from a network layer, ICMPv4 message values MAY be translated into ICMPv6 message values based on [SIIT], and vice versa. It can be implemented using raw socket.

6.3 IPv4 or IPv6 Address Pool and Mapping Table

The address pool consists of the unassigned IPv4 addresses or internal IPv6 addresses. This pool can be implemented at different granularity in the node e.g., a single pool per node, or at some finer granularity such as per user or per process. However, if a number of IPv4 applications communicate with IPv6 hosts or IPv6 applications communicate with IPv4 hosts, the available address spaces will be exhausted. As a result, it will be impossible for IPv4 applications to communicate with IPv6 nodes or IPv6 applications to communicate with IPv4 nodes. It requires smart management techniques for address pool. For example, it is desirable for the mapper to free the oldest entry and reuse the IPv4 address or IPv6 address for creating a new entry. This issues is the same as [BIS]. In case of a per-node address mapping table, it MAY cause a larger risk of running out of address.

6.4 Internally Assigned IPv4 or IPv6 Addresses

The IPv4 addresses, which are internally assigned to IPv6 target hosts out of the pool, are the unassigned IPv4 addresses (e.g., 0.0.0.1 ~ 0.0.0.255). There is no potential collision with another use of the private address space when the IPv4 address flows out from the host.

The IPv6 addresses, which are internally assigned to IPv4 target hosts out of the pool, are the special IPv6 addresses (It need IANA's consideration). There is no potential collision with another use of
thosee address space when the IPv6 address flows out from the host.

6.5 Mismatch between DNS result (AAAA) and Peer Application Version (v4)

If a server application you are using does not support IPv6 or IPv4 yet, but runs on a machine that supports other IPv6 or IPv4 services and this is listed with a AAAA or A record in the DNS, a client IPv4 or IPv6 application using BIA might fail to connect to the server application, because there is a mismatch between DNS query result (i.e., AAAA) and a server application version (i.e., IPv4, IPv6). A solution is to try all the addresses listed in the DNS and just not fail after the first attempt. We have two approaches: the client application itself SHOULD cycle through all the addresses and end up trying the IPv4 or IPv6 one. Or it SHOULD be done by some extensions of name resolver and API translator in BIA. For this, BIA SHOULD do iterated jobs for finding the working address used by the other application out of addresses returned by the extended name resolver. It may very well be application dependent. Note that BIA might be able to do the iteration over all addresses for TCP sockets, since BIA can observe when the connect call fails. But for UDP sockets it is hard if not impossible for BIA to know which address worked, hence the application must do the iteration over all addresses until it finds a working address.

Another way to avoid this type of problems is to make BIA only come into effect when no A records exist for the peer. Thus traffic from an application using BIA on a dual-stack host to a dual-stack host would use IPv4 or IPv6.

6.6 Implementation Issues

Some operating systems support the preload library functions, so it is easy to implement the API translator by using it. For example, the user can replace all existing socket API functions with user-defined socket API functions which translate the socket API function. In this case, every IPv4 application has its own translation library using a preloaded library which will be bound into the application before executing it dynamically.

Some other operating systems support the user-defined layered protocol allowing a user to develop some additional protocols and put them in the existing protocol stack. In this case, the API translator can be implemented as a layered protocol module.

In the above two approaches, it is assumed that there exists both TCP(UDP)/IPv4 and TCP(UDP)/IPv6 stacks and there is no need to modify or to add a new TCP-UDP/IPv6 stack.
7. Limitations

In common with [NAT-PT], BIA needs to translate IP addresses embedded in application layer protocols, e.g., FTP. So it may not work for new applications which embed addresses in payloads.

This mechanism supports unicast communications only. In order to support multicast functions, some other additional functionalities must be considered in the function mapper module.

Since the IPv6 API has new advanced features, it is difficult to translate such kinds of IPv6 APIs into IPv4 APIs. Thus, IPv6 inbound communication with advanced features may be discarded.

8. Security Considerations

The security consideration of BIA mostly relies on that of [NAT-PT]. The differences are due to the address translation occurring at the API and not in the network layer. That is, since the mechanism uses the API translator at the socket API level, hosts can utilize the security of the network layer (e.g., IPsec) when they communicate with IPv6 or IPv4 hosts using IPv4 or IPv6 applications via the mechanism. As well, there isn’t a DNS ALG as in NAT-PT, so there is no interference with DNSSEC.

The use of address pooling may open a denial of service attack vulnerability. So BIA should employ the same sort of protection techniques as [NAT-PT] does.

9. Acknowledgments

The authors gratefully acknowledge the many helpful advice from Dan Wing and Dave Thaler for initiating this work, thanks mailing list discussion from Mohamed Boucadair, Yiu L. Lee, James Woodyatt, Lorenzo Colitti, Qibo Niu and Pierrick Seite. Contributions from Gang Chen, Bo Zhou, Dapeng Liu, Hong Liu, Tao Sun et al. in the development of this document.

10. References


[SIIIT-NEW] Li, X., Bao, C., Baker, F., "IP/ICMP Translation Algorithm", draft-ietf-behave-v6v4-xlate-01, September

Huang, et al. Expires April 22, 2010 [Page 22]
2009, work-in-progress


[DNS64] Bagnulo, M., Sullivan, A., Matthews, P., van Beijnum, I., "DNS64: DNS extensions for Network Address Translation from IPv6 Clients to IPv4 Servers", draft-ietf-behave-dns64-00, July 2009, work-in-progress


Appendix A: API list intercepted by BIA

The following functions are the API list which SHOULD be intercepted by BIA module.

The functions that the application uses to pass addresses into the system are:

- bind()
- connect()
- sendmsg()
- sendto()

The functions that return an address from the system to an application are:

- accept()
- recvfrom()
- recvmsg()
- getpeername()
- getsockname()

The functions that are related to socket options are:

- getsockopt()
- setsockopt()

The functions that are used for conversion of IP addresses embedded in application layer protocol (e.g., FTP, DNS, etc.) are:

- recv()
- send()
- read()
- write()

As well, raw sockets for IPv4 and IPv6 MAY be intercepted.

Most of the socket functions require a pointer to the socket address structure as an argument. Each IPv4 argument is mapped into corresponding an IPv6 argument, and vice versa.

According to [SOCK-EXT], the following new IPv6 basic APIs and structures are required.
BIA MAY intercept inet_ntoa() and inet_addr() and use the address mapper for those. Doing that enables BIA to support literal IP addresses.

The gethostbyname() call return a list of addresses. When the name resolver function invokes getaddrinfo() and getaddrinfo() returns multiple IP addresses, whether IPv4 or IPv6, they SHOULD all be represented in the addresses returned by gethostbyname(). Thus if getaddrinfo() returns multiple IPv6 addresses, this implies that multiple address mappings will be created; one for each IPv6 address.
Authors’ Addresses

Bill Huang  
China Mobile  
53A, Xibianmennei Ave.,  
Xuanwu District,  
Beijing 100053  
China  

Email: bill.huang@chinamobile.com

Hui Deng  
China Mobile  
53A, Xibianmennei Ave.,  
Xuanwu District,  
Beijing 100053  
China  

Email: denghui02@gmail.com

Teemu Savolainen  
Nokia  
Hermiankatu 12 D  
FI-33720 TAMPERE  
Finland  

Email: teemu.savolainen@nokia.com