ISSN:2249-5789

NAT Traversal Capability and Keep-Alive Functionality with IPSec in IKEv2 Implementation

CHAMAN SINGH[1] K.L.BANSAL[2]

[1]*Research Scholar* [2]*Associate Professor*

chaman83mca@gmail.com *kishorilalbansal@yahoo.co.in*

Department of Computer Science, Himachal Pradesh University Shimla, India

Abstract

Since IPv4 Private Networks are behind NAT (Network Address Translation) devices. So, to bypass the Binding Update and Binding Acknowledgment by NAT, we need to encapsulate it in UDP (User datagram Protocol) Packets. Hence, the Dual Stack Mobile IPv6 should support NAT Traversal and Detection. So for proper securing and fully functionality of NAT traversal, it should be IP Security Protected. Paper presents design and implementation of NAT traversal capability and keeps alive functionality with IP Security in IKEv2 (Internet Key Exchange version 2) implementation for proper Data Communication. It also implements how IPSec integrate with NAT.

Keywords-*Network Address Translation, Traversal, Detection, IP Security, Home Link, Data Traffic, Linux Kernel, IKEv2.*

1. Introduction

The Mobile IPv6 [1] is a protocol developed as a subset of Internet Protocol veMyon 6[2] to support mobile connections. Mobile IPv6 allows a mobile node to transparently maintain connections while moving from one [3] subnet to another [4]. The Mobile IPv6 protocol takes care of binding addresses between Home Agent and Mobile Node. It also ensures that the Mobile Node is always reachable through Home Agent. Dual Stack Mobile IPv6 [5] is an extension of MIPv6 to support mobility of devices irrespective of IPv4 and IPv6 network. NEPL (NEMO Platform for Linux) [6] is a freely available implementation of DSMIPv6 for Linux platform. The original NEPL release was based on MIPL (Mobile IPv6 for Linux) [7]. In DSMIPv6, all Mobile Nodes has a fixed address, called a Home Address assigned by Home Agent. When the MN moves to other networks, it gets Care-of Address from foreign network. MN sends a Binding Update message to its home agent. Then Home Agent replies to the Mobile Node with a Binding Acknowledgement message to confirm the request. When MN is moved to any foreign network all packets sent to the Home agent will be IPSec encrypted. A bi-directional tunnel [8] is established between the Home Agent and the care of address of the Mobile Node after the binding information has been successfully exchanged. DSMIPv6 [9] extends the Mobile IPv6 and NEMO Basic Support standards to allow Mobile Nods to roam in both IPv6 and IPv4-only networks. [9] Solution is an extension to the existing NEPL solution provided by Nautilus [10]. Network Address Translation (NAT) [15] was meant to be temporary, but it's now in widespread use and it's actually holding back wider deployment of IPv6. Apart from the address shortage, Internet also has security related problems. There are different solutions for these problems currently in use, of which we are particularly interested in IPsec. IPsec is architecture [16], currently in a second generation that defines behaviour of compliant IPsec nodes. Those are Encapsulating Security Payload (ESP) [17] used for traffic encryption and integrity protection, and Authentication Header (AH) [18]. Of those two, ESP is mandatory to implement, while AH was mandatory, but now is optional. The third protocol is Internet Key

ISSN:2249-5789

Exchange version 2 (IKEv2) [19] and it is used for authentication, authorization and key exchange within IPsec (Internet Protocol Security) architecture. Widespread deployment of NAT based devices creates substantial problems to IPsec protocols. As we implemented NAT in IKEv2 protocol we had to do thorough analysis of possible problems and their solutions. This paper summarizes our design and code changes for proper communication in IKEv2 implementation. MY validated the DSMIPv6 functionality as per the requirements provided against the draft-ietf-mext-nemo-v4traversal-08.txt I-D, [5] along with other IETF standards. I have taken baseline architecture implementation from the Nautilus6 which uses Linux platform [11].

2. Network Address Translation

NAT (Network Address Translation) [13] is the translation of an Internet Protocol address (IP address) used within one network to a different IP address known within another network. One network is designated the inside network and the other is the outside. In DSMIPv6 the mip6d Daemon should bypass NAT, when MN is behind Nat' Ted device in IPv4 FL.

NATs were introduced primarily because of the shortage of IPv4 addresses. IP nodes that are "behind" a NAT device have IP addresses that are not globally unique. They are more often assigned from some space that is unique within the network behind the NAT but which are likely to be reused by nodes behind other NATs. Node behind a NAT, which wants to communicate with other node on the Internet, is assigned a global IP address by NAT box which results with change of source IP address for outgoing packets. Similar situation is when destination node is behind a NAT, then for incoming packets NAT box changes destination IP address to the private IP address of node on the internal network. NAT box keeps the mapping for the duration of the communication. This duration is estimated by NAT box heuristically. Mapping is often achieved by additional translation based on UDP or TCP ports. In that case, NAT box is known as a NAPT box. There are many protocols having complications with NAT [20]. Applications such as FTP, H.323, SIP and RTSP use a control connection to establish a data flow and they are usually broken by NAT devices en-route. This is because these applications exchange address and port parameters within control session to establish data session and session orientations.

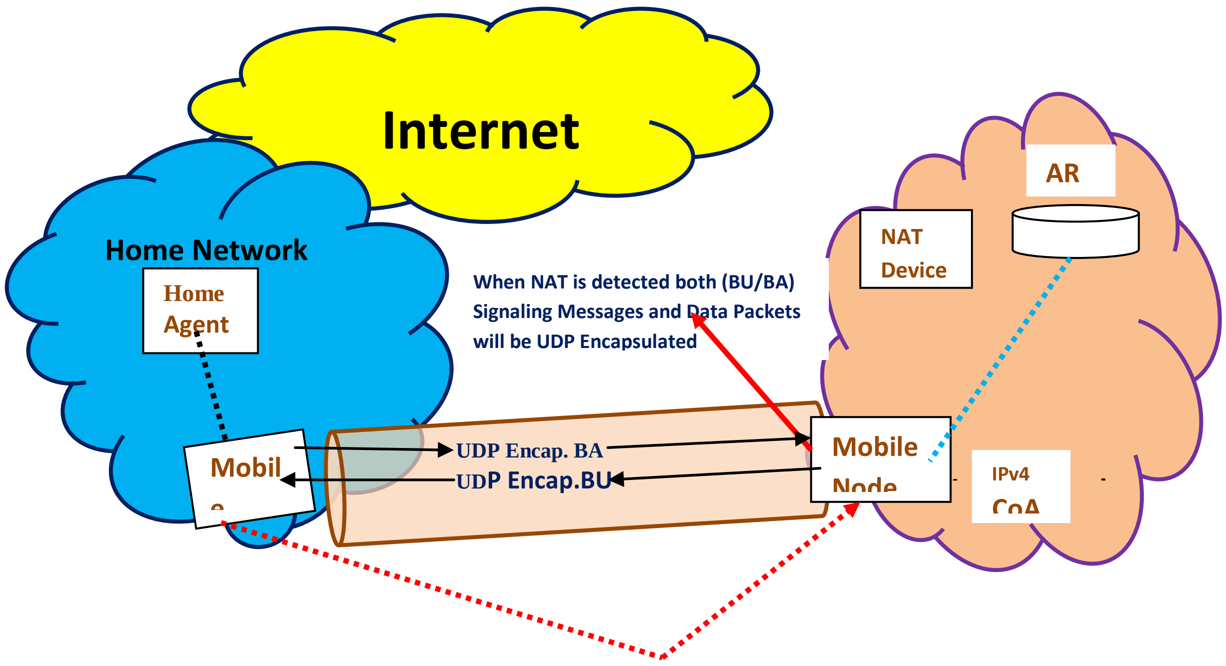
Internet
Home Network
Home Agent
Mobil e
When NAT is detected both (BU/BA) Signaling Messages and Data Packets will be UDP Encapsulated
UDP Encap. BA
UDP Encap.BU
NAT Device
AR
Mobile Node
IPv4 CoA

Figure 1: NAT Detection and Traversal Module .

Most likely reasons for failures are that addressing information in payload could be realm specific and second, that control sessions permit data sessions to originate in a direction that NAT might not permit. Peer to peer applications also have problems with NAT. They can be originated by any of the peers and external peers will not be able to locate their peers in private realm unless they know the externally assigned IP address. Applications requiring retention of address mapping or requiring more public addresses than available are broken by NAT for obvious reasons. Namely, in the first case NAT cannot know this requirement and may assign external addresses between sessions to different hosts and in the second case NAT is limited by number of available public addresses.

3. NAT Traversal and Detection Design

NAT Detection [14] is done when the initial Binding Update message is sent from the mobile node to the home agent. When located in an IPv4-only foreign link, the mobile node sends the Binding Update message encapsulated in UDP and IPv4, this is handled in a particular file. When the home agent receives the encapsulated Binding Update, it compares the IPv4 address of the source address field in the IPv4 header with the IPv4 address included in the IPv4 care-of address option. Otherwise, a NAT is detected in the path and the NAT detection option is included in the Binding Acknowledgement. The Binding Acknowledgement, and all future packets, is then encapsulated in UDP and IPv4. Note that the home agent also stores the port numbers and associates them with the mobile node's tunnel in order to forward future packets. The mip6d Daemon adds the xfrm polices/states for UDP encapsulation of BA and IPv6/IPv4 data traffic [21]. Upon receiving the Binding Acknowledgement with the NAT detection option, the mobile node sets the tunnel to the home agent for UDP encapsulation. Hence, all future packets to the home agent are tunnelled in UDP and IPv4. If no NAT device was detected in the path between the mobile node and the home agent then IPv4/IPv6 data traffic is not UDP encapsulated. A mobile node will always tunnel the Binding Updates in UDP when located in an IPv4-only network. Essentially, this process allows for perpetual NAT detection. Similarly, the home agent will encapsulate Binding Acknowledgements in a UDP header whenever the Binding Update is encapsulated in UDP. The mip6d Daemon adds xfrm polices/states for UDP encapsulation of IPv6/IPv4 data traffic, when NAT was detected between MN and HA.

4. IPSec for Private Networks

IPsec keeps records about traffic which needs to be protected and how to protect it in two databases – SPD (Security Policy Database) and SAD (Security Association Database). SPD contains entries about security policy – which traffic to protect, which protocol to use, level of protection etc. Traffic selectors specify which packets to protect by specifying source and destination addresses, upper layer protocols and ports. IPsec is based on SA (Security association), which is a set of security parameters, for instance crypto algorithms used in communication. SA is uniquely defined by protocol (AH or ESP), destination IP address and SPI (Security Parameters Index). Two sides will establish connection if and only if they successfully negotiate security parameters for the connection. ESP and AH are two main security protocols in the Ipsec architecture which assure traffic protection. AH is used for authentication and integrity check, while ESP is used primary to enable confidentiality and optionally, authentication and integrity check. There are two IPsec modes: transport mode and tunnel mode. Transport mode is appropriate for usage when communication is end-to-end. In this mode, we have only one source and destination IPv4 address, which are in AH protected by ICV. This leads to problems with NAT, as described later. ESP doesn't have these problems because his integrity check doesn't cover IPv4 header where are these addresses situated. Tunnel mode is better where communication takes place between security gateways. In this case communication is maintained within the Ipsec tunnel. This leads to another pair of IP addresses and therefore, to another header besides the original one: "outer" IP header. ESP encryption now covers whole IPv4 datagram, with inner header also. AH authentication checks integrity of the both inner and outer IPv4 header, and off course, IPv4 payload. Integrity check successfully reveals attempts of packet change by intruder on insecure network. AH and ESP require cryptographic keys to be in SA database. Though possible manual key management isn't particularly secure and doesn't scale well. These problems are solved by automatic key exchange, specifically by Internet Key Exchange version 2 (IKEv2) protocol. Daemon, which runs IKEv2 protocol, generates symmetric keys and does rekeying after some period. Authentication in IPsec is also performed by the IKEv2 protocol using pre-shared keys, digital certificates or EAP. IKEv2 messages are transferred via UDP protocol in pairs, requests and response. Each pair is known as *exchange*. Communication between two IKEv2 entities is established via two exchanges. Establishment of SA includes traffic selectors and cryptographic algorithms to use for data protection. There are two design possibilities with the respect to interrelation of IPsec and NAT devices. The first one is for IPsec protocols to completely ignore NAT, while the other one is to introduce mechanisms in the protocol that will allow IPsec compliant devices to communicate in spite of NAT devices.

5. Interaction of IPSec and IKEv2

XFRM [11] is a packet transformation framework residing in the Linux kernel. It performs operations on IP packets such as inserting, modifying headers, UDP encapsulation and de-capsulation. DSMIPv6 XFRM module will take the advantage of existing IPSEC transformation and defines a simple UDP encapsulation scheme. IPSEC module is responsible for interaction with IKEV2 through MIGRATE messages. IPSec will be used to protect the following traffic between Home Agent and Mobile Node.

1. BU/BA messages.
2. Mobile prefix sollicitation and advertisement messages.
3. Normal traffic between Mobile Node and Home Agent.
4. All tunneled normal traffic between Mobile Node and correspondent Node.

In Mip6d, the Mobile Node (MN) and the Home Agent (HA) uses IPsec Security Associations (SAs) in transport mode to protect BU/BA messages, since the MN may change its attachment point to the Internet, it is necessary to update its endpoint address of the IPsec SAs.
This indicates that corresponding entry in IPsec databases (Security Policy (SPD) and SA (SAD) databases) should be updated when Mobile Node performs movements. IPSec is used to protect the following traffic between Home Agent and Mobile Node:

BU/BA messages: IPSec Protection for BU/BA

When Mobile Node moves in FL a new Care of address is assigned to the Mobile Node by FL network. After detecting the movement following steps are taken to create IPSec tunnel.

1. Mip6d issues a PF_KEY MIGRATE message to the PF_KEY socket.
2. The operating system validates the message and checks if corresponding security policy entry exists in SPD.
3. When the message is confirmed to be valid, the target SPD entry is updated according to the MIGRATE message. If there is any target SA found that are also target of the update, those should also be updated.

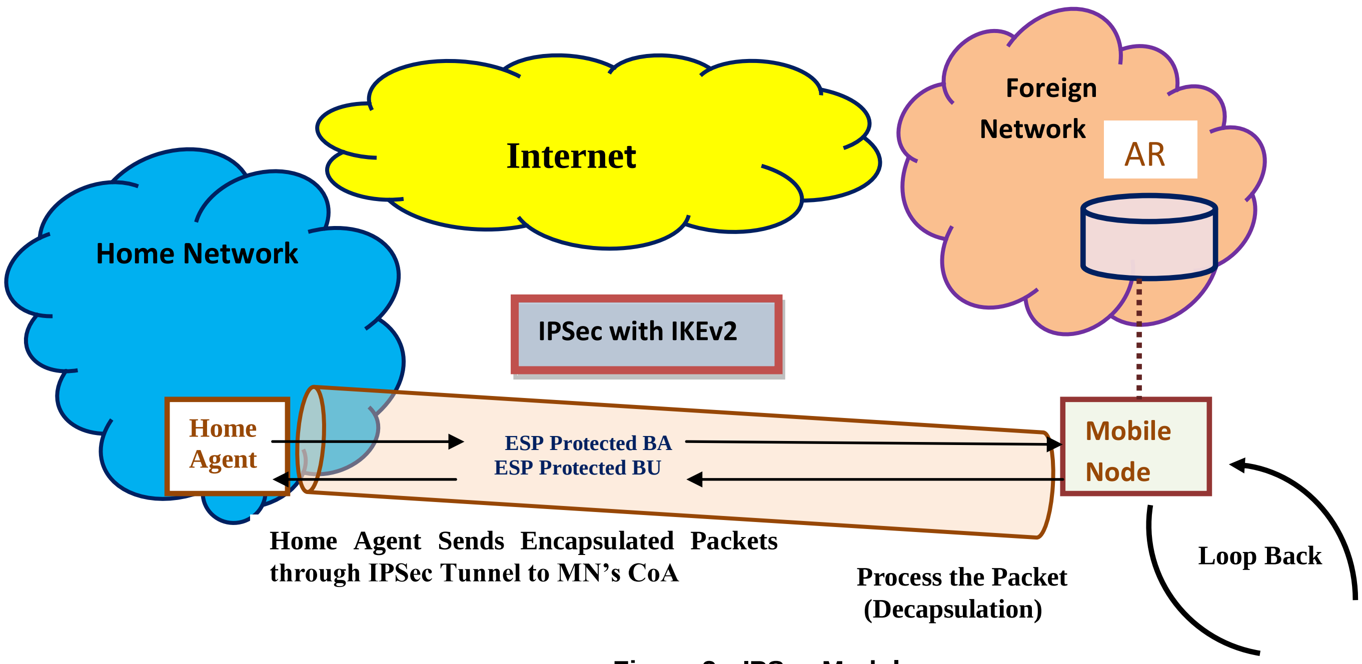

Figure 2:- IPSec Module

4. After the MIGRATE message is successfully processed inside the kernel, it will be sent to all open PF_KEY sockets. The IKE daemon receives the MIGRATE message from its PF_KEY socket and updates its SPD and SAD images. The IKE daemon may also update its state to keep the IKE session alive.

5. After that ESP protected BU is send with K–bit set.

Mobile IPv6 specifies a flag named Key Management Mobility Capability bit (K-bit) in Binding Update (BU) and Binding Acknowledgement (BA) messages, which indicates the ability of IKE sessions to survive movement. When both the Mobile Node and Home Agent agree to use this functionality, the IKE daemons dynamically update the IKE session when the Mobile Node moves.

6. Description and Implementation

It contains the details about patches applied and code changes done by me in Linux kernel, mipv6 Daemon and strongSwan in different releases of DSMIPv6. The scope of this release is to demonstrate the following working scenarios:

Scenario 1: Movement of MN from HL to IPv4 network.

Scenario 2: Movement of MN from IPv4 to HL network.

Scenario 3: Movement of MN from IPv6 to IPv4 network.

Scenario 4: Movement of MN from IPv4 to IPv6 network.

It also captures the working of the below mentioned features to demonstrate the above mentioned scenarios:

1. Security considerations related to IPV6 with IPSEC and IKEv2.
2. Handover interactions for IPSec and IKE
3. IKE negotiations between MN and HA.
4. IKEv2 operation for securing DSMIPv6 signalling (BU & BA).
5. NAT Detection in MN and HA.
6. NAT Traversal in MN and HA
7. UDP Encapsulation of signalling (BU/BA) Messages.
8. UDP Encapsulation of IPv6 and IPv4 data traffic.

6.1 Linux Kernel

Description**:** Used patched Linux kernel 2.6.28.2 for this release.

VeMyon: 2.6.28.2

Tar file**:** linux-2.6.28.2.tgz

6.2 User land-DSMIP

Description: User land DSMIP Daemon used in mipv6 taken from nautilus site. All patches applied to support DSMIPv6.

VeMyon: 0.4

Tar file**:** mipv6-Daemon-umip-0.4.tgz

6.3 User land IKEv2 Daemon

Description: Used strongSwan package as user land IKEv2 Daemon

VeMyon: 4.2.9

Tar file**:** strongswan-4.2.9.tgz

6.4 Changes done in user land-DSMIP

Description: Code changes have been made in mip6d Daemon for successfully detecting NAT.

6.4.1 Change list 1.

Bug Description: NAT Detection logic was failing, when we move from IPv6 FL to IPv4 FL.
File Modified: ha.c
Function Modified: ha_recv_bu_worker
Bug Fix Description: NAT detection logic was failing, when we move from IPv6 to IPv4 FL. In this scenario the CoA in bce (Binding Cache Entry) was old value that is IPv6 Address. Due to which IP addresses (source IP & CoA in bce) were getting mismatched and NAT was getting detected, which is wrong behavior. So at the time of comparing addresses, using CoA from out.bind_coa instead of CoA in BCE...
Code Snippet for Minor Changes

NAT detection logic was failing, when we move from IPv6 to IPv4. In this scenario the CoA in bce is old value that is IPv6 Address. Due to which IP addresses (src IP & CoA in bce) were getting mismatched and NAT was getting detected, which is wrong behaviour. So at the time of comparing addresses, using CoA from out.bind_coa instead of CoA in BCE

```
If (! IN6_ARE_ADDR_EQUAL
(&v4mapped_src, out.bind_coa))

    {
            Bce->behind_nat = 1;

        }
```

6.4.2 Change list 2.

Bug Description: At the time of movement from IPv4 to IPv6 Network (or HL to IPv6). Sometime sit device was going in wrong state. And if after that MN moves from IPv6 to IPv4. Mip6d was not able to access the sit device and finally deleted the sit interface. As there was no sit device in MN, mip6d code throws assertion, when it tries to modify the sit tunnel endpoints.

File Modified: mn.c

*Function Modified: mn*_tnl_state_add

New Functions Added:

mn_clr_interface_flag

mn_set_interface_flag

index2name

Bug Fix Description: To avoid the assertion currently added the hack to avoid this scenario. Doing down or up of sit device, only when MN moves from IPv4toIPv6 FL or HLtoIPv6 FL. Latter need some better fix.

Comparison of Files for Major Changes: Comparison of changes done to fix the issue in mn.c

6.4.3 Change list 3.

Bug Description: When MN moves second time from HL to IPv4, the IPv6 HoA is assigned to ip6tnl and latter it should move from ip6tnl to sit device, which was not happening. The correct behavior should be same as we do for IPv4 HoA. At the first movement from HL to IPv4, sit device is assigned IPv6 HoA correctly, because hai to if_tunnel stores the sit index value initially.

File Modified: mn.c

Function Modified: process_first_home_bu

Bug Fix Description: When Mobile Node moves second time from Home Link to IPv4 FL, moving IPv6 HoA from ip6tnl to sit device, by calling routine mv_hoa

Code Snippet for Minor Changes: When MN moves second time from HL to IPv4, the IPv6 HoA is assigned to ip6tnl and latter it should move from ip6tnl to sit device, which was not happening. The correct behavior should be same as we do for IPv4 HoA. At the first movement from HL to IPv4, sit device is assigned IPv6 HoA correctly, because ha to is_tunnel * stores the sit index value. Done changes to fix this issue....

```
        if (hai->if_tunnel!= hai->if_tunnel64)
        {           struct mv_hoa_args mha;
                    mha.if_next        =        hai-
>if_tunnel64;
                    mha.target = hai;
                    addr_do (&hai->hoa.addr, 128,
        hai->if_tunnel, &mha, mv_hoa);
          }
         hai->if_tunnel = hai->if_tunnel64;
```

6.4.4 Change list 4.

Bug Description:

There was bug in mipv6 logic; it was not taking the prefix length of IPv4 HoA configured by user in configuration file. Due to which problem was coming in setting v4 route, when MN boots in IPv6 FL and moves to IPv4 FL.
File Modified: mn.c

Function Modified: flag_hoa4

Bug Fix Description: Done code changes to correct the logic, so that mipv6 Daemon should take the prefix length of IPv4 HoA, if configured by user in mip6d configuration file.

Code Snippet for Minor Changes: Changes have been made to accept the actual prefix Length from the configuration file. This case occurs when we boot the MN in IPV6 LINK, here ifa_index if_tunnel4 are different so take the prefix length from conf file. This also solves when MN moves IPV6 to IPV4 LINK because ifa_index and hai_index are not equal so it will take 32 as prefix length. This resolves the v4 route issue when MN boots up in IPV6 link.

```
 plen4 = (ifa->ifa_index! = hai->if_tunnel4? 32:
hai->plen4);
```

6.4.5 Change list 5.

Bug Description: IPv4 traffic was not passing through tunnel device in IPv4 FL

File Modified: dhcp_dna.c, dhcp_dna.h, mn.c

Function Modified: dhcp_configuration

New Functions Added:

mn_coa_route_add,
mn_route_coa_del

Bug Fix Description:

Added source based route to pass IPv4 traffic through tunnel device and not through egress interface of MN directly.

Code Snippet for Minor Changes:

dhcp_dna.h

Structure for storing the route in formation needed in IPV4 link

```
struct dhcp_route
{
        int if_index;
        unsigned long gateway;
};
```

dhcp_dna.c

Copying the value of Gateway and Ifindex value to dhcp_route structure Will needed in creating a source based route for BU.

```
route_dhcp.if_index = if_index;
route_dhcp.gateway = dhcp_ctrl->gateway;
```

Comparison of files for Major Changes: Comparison of changes done to fix the issue in mn.c

6.4.6 Change list 6.

Bug description: External IPv4 network of Home Agent was not reachable from MN in HL.

File Modified: mn.c

Function Modified: mn_move

Bug Fix Description: - Adding default route to Home Agent IPv4 address, when Mobile Node is in HL. So that MN can reach to other network than HL. Route is added on the physical interface where IPv4 HoA is configured.

Code Snippet for Minor Changes: Adding default route to HOMAGENTV4ADDRESS when MN is in HOME LINK So that Mobile Node can reach to other network than HL.Route is added on the physical interface where IPv4 Home Address is configured.

```
MDBG ("Default route is added in HL toward iface_index %d\n", hai->hoa.iif);
if (route4_add(hai->hoa.iif, RT6_TABLE_MAIN, NULL , NULL, 0, &any4, 0, &ha4_addr) < 0)
MDBG ("Default route insertion failed for MN in HL.\n");
CHANGES: Deletion of default route to HOMAGENT V4ADDRESS, when MN is not in HOMELINK.
 MDBG ("Default route is deleted in FL toward iface_index %d\n",hai->hoa.iif);
if (route4_del (hai->hoa.iif, RT6_TABLE_MAIN,NULL, 0, &any4,0,&ha4_addr) < 0) MDBG("Default route deletion failed for MN in HL.\n");
```

6.4.7 Change list.

Bug Description: When MN was behind NAT in IPv4 FL, the large size IPv4/IPv6 data traffic was not getting exchanged. After initial handshake, client was not able to exchange data traffic and was in hang state. In nutshell everything that uses large packets was not working.

File Modified: - tunnelctl.c

Function modified: __tunnel44_add

__tunnel64_add

Bug Fix description: Set the MTU size of tunnel device to 1472 instead of 1480, left 8 bytes for UDP Encapsulation header.

Code Snippet for Minor Changes: **__tunnel44_add**

```
[DSMIP_BUG]: When MN was behind NAT in IPv4 FL, the large size IPv4/IPv6 data traffic was not getting exchanged. After initial handshake, client was not able to exchange data traffic and was in hang state. In nutshell everything that uses large packets was not working. Fix: set the MTU size of tunnel device to 1472 instead of 1480, left 8 bytes for UDP Encapsulation header...
   ifr.ifr_ifru.ifru_mtu = MAX_MTU_SIZE_FOR_TUNL_DEV;
   if (ioctl(tnl44_fd, SIOCSIFMTU, &ifr) < 0)
{
       TDBG ("SIOCSIFFLAGS failed MTU %d
%s\n",
       errno, strerror(errno));
       goto err;
  }
```

__tunnel64_add

CHANGES: [DSMIP_BUG]: When MN was behind NAT in IPv4 FL, the large size IPv4/IPv6 data traffic was not getting exchanged. After initial handshake, client was not able to exchange data traffic and was in hang state. In nutshell everything that uses large packets was not working. Fix: set the MTU size of sit device to 1472 instead of 1480, left 8 bytes for UDP Encapsulation header. ifr.ifr_ifru.ifru_mtu=MAX_MTU_SIZE_FOR_SIT _DEV;   if (ioctl (tnl4_fd, SIOCSIFMTU, &ifr) < 0)

```
 {
           TDBG ("SIOCSIFFLAGS failed MTU %d
%s\n",
           errno, strerror (errno));
           goto err;
  }
```

6.5 Changes done in user land-DSMIP

6.5.1 Changes done in user land IKEv2 Daemon.

Description: Integration of IPsec with NAT
Change list-1:
Bug Description: IPsec signalling when MN was behind NAT was failing.

File Modified:

strongswan-4.2.9/src/charon/sa/tasks/ child_create.c

strongswan-4.2.9/src/charon/plugins/kernel_netlink/ kernel_netlink_ipsec.c

Function Modified: select_and_install, status_t add_sa

Code Snippet: **child_create.c:** This code is present in function **select_and_install ()** of   if (! this->initiator)

{   /* check if requested mode is acceptable, down grade if required */

```
switch (this->mode)
  {
          case MODE_TRANSPORT:

   if (!this->config->use_proxy_mode(this->config) &&(!ts_list_is_host(this->tsi, other) ||
                                     !ts_list_is_host(this->tsr, me)))
                                    {
                                                this->mode = MODE_TUNNEL;
                                                DBG1(DBG_IKE, "not using transport mode, not host- to-host");
                                     }
             else if (this->ike_sa->has condition (this->ike_sa,
             COND_NAT_ANY))
              {
                          /* Do not switch to tunnel mode when nat is detected. For securing signaling we
                          need       transport mode SA as per draft. So stopping switches to tunnel mode in
                          case of NAT-T. DBG1 (DBG_IKE, "not using transport mode, connection
                          Nated"); */
                         //this->mode = MODE_TUNNEL;
                        DBG1 (DBG_IKE, "using transport mode, connection NATed");
               }
                break;
               case MODE_BEET:
                if (!ts_list_is_host(this->tsi, NULL) ||
                       !ts_list_is_host(this->tsr, NULL))
                {
                       this->mode = MODE_TUNNEL;
                       DBG1 (DBG_IKE, "not using BEET mode, not host-to-
                       host");
                }
```

Figure 3:- Implementation of Kernel Interface.

```
                        break;
                default:
                        break;
            }
  }
```

kernel_netlink_ipsec.c

```
 /*Implementation of kernel_interface_t.add_sa. */
static status_t add_sa (private_kernel_netlink_ipsec_t *this, host_t *src, host_t *dst, u_int32_t spi,
protocol_id_t protocol,  u_int32_t reqid,u_int64_t expire_soft, u_int64_t expire_hard, u_int16_t enc_alg, chunk_t
enc_key, u_int16_t int_alg, chunk_t int_key,ipsec_mode_t mode, u_int16_t ipcomp,  u_int16_t cpi,  bool encap,
bool inbound)
{
        netlink_buf_t request;
        char *alg_name;
        struct nlmsghdr *hdr;
        struct xfrm_usersa_info *sa;
        u_int16_t icv_size = 64;
        /* if IPComp is used, we install an additional IPComp SA. if the cpi  is 0 we are in the recuMyve call below
*/
        if (ipcomp != IPCOMP_NONE && cpi != 0)
        {
             add_sa(this, src, dst, htonl(ntohs(cpi)), IPPROTO_COMP, reqid,
            0,0,ENCR_UNDEFINED,chunk_empty,     AUTH_UNDEFINED,chunk_empty, mode, ipcomp, 0,
            FALSE, inbound);
             ipcomp = IPCOMP_NONE;
        }
        memset(&request, 0, sizeof(request));
        DBG2(DBG_KNL, "adding SAD entry with SPI %.8x and reqid{%u}", ntohl(spi), reqid);
        hdr = (struct nlmsghdr*)request;
        hdr->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
        hdr->nlmsg_type = inbound ? XFRM_MSG_UPDSA
```

Figure 4:- Implementation of Kernel Interface.

```
      /* we currently do not expire SAs by
       volume/packet  count */
       sa->lft.soft_byte_limit = XFRM_INF;
       sa->lft.hard_byte_limit = XFRM_INF;
       sa->lft.soft_packet_limit = XFRM_INF;
       sa->lft.hard_packet_limit = XFRM_INF;
       /* we use lifetimes since added, not since used
*/
       sa->lft.soft_add_expires_seconds =
expire_soft;
       sa->lft.hard_add_expires_seconds =
expire_hard;
       sa->lft.soft_use_expires_seconds = 0;
sa->lft.hard_use_expires_seconds = 0;
struct rtattr *rthdr = XFRM_RTA(hdr, struct
xfrm_usersa_info);
switch (enc_alg)
{
```

```
        case ENCR_UNDEFINED:
                /* no encryption */
                break;
        case ENCR_AES_CCM_ICV16:
        case ENCR_AES_GCM_ICV16:
                icv_size += 32;
                /* FALL */
        case ENCR_AES_CCM_ICV12:
        case ENCR_AES_GCM_ICV12:
                icv_size += 32;
                /* FALL */
        case ENCR_AES_CCM_ICV8:
        case ENCR_AES_GCM_ICV8:
        {
                rthdr->rta_type =
XFRMA_ALG_AEAD;
                alg_name =
lookup_algorithm (encryption_algs, enc_alg);
if (alg_name == NULL)
               {
                                             DBG1(DBG_KNL, "algorithm %N not supported by kernel!",
                                              encryption_algorithm_names, enc_alg);
                                              return FAILED;
               }
               DBG2(DBG_KNL, "  using encryption algorithm %N with key size %d",
               encryption_algorithm_names, enc_alg, enc_key.len * 8);
               rthdr->rta_len = RTA_LENGTH(sizeof(struct xfrm_algo_aead) +  enc_key.len);
               hdr->nlmsg_len += rthdr->rta_len;
               if (hdr->nlmsg_len > sizeof(request))
               {
                                                 return FAILED;
               }
               struct xfrm_algo_aead* algo = (struct  xfrm_algo_aead*)RTA_DATA(rthdr);
               algo->alg_key_len = enc_key.len * 8;
               algo->alg_icv_len = icv_size;
               strcpy(algo->alg_name, alg_name);
               memcpy(algo->alg_key, enc_key.ptr, enc_key.len);
               rthdr = XFRM_RTA_NEXT(rthdr);
               break;
        }
       default:
       {
               rthdr->rta_type = XFRMA_ALG_CRYPT;
               alg_name = lookup_algorithm(encryption_algs, enc_alg);
               if (alg_name == NULL)
               {
                                  DBG1(DBG_KNL, "algorithm %N not supported by kernel!",
                                               encryption_algorithm_names, enc_alg);
                                              return FAILED;
               }
               DBG2(DBG_KNL, "  using encryption algorithm %N with
            key size  %d",  encryption_algorithm_names, enc_alg, enc_key.len * 8);
           rthdr->rta_len = RTA_LENGTH(sizeof(struct xfrm_algo) + enc_key.len);
           hdr->nlmsg_len += rthdr->rta_len;
               if (hdr->nlmsg_len > sizeof(request))
               {
                      return FAILED;
               }
               struct xfrm_algo* algo = (struct  xfrm_algo*)RTA_DATA(rthdr);
               algo->alg_key_len = enc_key.len * 8;
               strcpy(algo->alg_name, alg_name);
               memcpy(algo->alg_key, enc_key.ptr, enc_key.len);
               rthdr = XFRM_RTA_NEXT(rthdr);
               break;
        }
 }
if (int_alg  != AUTH_UNDEFINED)
 {
        rthdr->rta_type = XFRMA_ALG_AUTH;
       alg_name = lookup_algorithm(integrity_algs, int_alg);
       if (alg_name == NULL)
       {
             DBG1(DBG_KNL, "algorithm %N not supported by  kernel!",integrity_algorithm_names, int_alg);
             return FAILED;
       }
```

Figure 5:- Implementation of Kernel Interface.

```
            DBG2(DBG_KNL, "  using integrity algorithm %N with key size %d",
            integrity_algorithm_names, int_alg, int_key.len * 8);
            rthdr->rta_len = RTA_LENGTH(sizeof(struct xfrm_algo) +  int_key.len);
            hdr->nlmsg_len += rthdr->rta_len;
            if (hdr->nlmsg_len > sizeof(request))
            {
                  return FAILED;
            }
            struct xfrm_algo* algo = (struct xfrm_algo*)RTA_DATA(rthdr);
            algo->alg_key_len = int_key.len * 8;
            strcpy (algo->alg_name, alg_name);
            memcpy (algo->alg_key, int_key.ptr, int_key.len);
            rthdr = XFRM_RTA_NEXT (rthdr);
      }
      if (ipcomp != IPCOMP_NONE)
      {
            rthdr->rta_type = XFRMA_ALG_COMP;
            alg_name = lookup_algorithm (compression_algs, ipcomp);
            if (alg_name == NULL)
            {
                  DBG1(DBG_KNL, "algorithm %N not supported by  kernel!",ipcomp_transform_names, ipcomp);
                  return FAILED;
            }
            DBG2(DBG_KNL, "  using compression algorithm %N", ipcomp_transform_names, ipcomp);
            rthdr->rta_len = RTA_LENGTH(sizeof(struct xfrm_algo));
            hdr->nlmsg_len += rthdr->rta_len;
            if (hdr->nlmsg_len > sizeof(request))
            {
                  return FAILED;
            }
            struct xfrm_algo* algo = (struct xfrm_algo*)RTA_DATA(rthdr);
            algo->alg_key_len = 0;
            strcpy(algo->alg_name, alg_name);
            rthdr = XFRM_RTA_NEXT(rthdr);
      }
IPsec protection for data packets is not needed as of Now. This piece of code is conflicting with mip6d.
     #if 0
      if (encap)
      {
            rthdr->rta_type = XFRMA_ENCAP;
            rthdr->rta_len = RTA_LENGTH(sizeof(struct
            xfrm_encap_tmpl));
            hdr->nlmsg_len += rthdr->rta_len;
            if (hdr->nlmsg_len > sizeof(request))
            {
                  return FAILED;
            }
            struct xfrm_encap_tmpl* tmpl = (struct
            xfrm_encap_tmpl*) RTA_DATA (rthdr);
            tmpl->encap_type = UDP_ENCAP_ESPINUDP;
            tmpl->encap_sport = htons(src->get_port(src));
            tmpl->encap_dport = htons(dst->get_port(dst));
            memset (&tmpl->encap_oa, 0, sizeof (xfrm_address_t));

            rthdr = XFRM_RTA_NEXT (rthdr);
      }
      #endif
      if (this->socket_xfrm->send_ack (this->socket_xfrm, hdr) !=
      SUCCESS)
      {
             DBG1 (DBG_KNL, "unable to add SAD entry with SPI %.8x", ntohl(spi));
            return FAILED;
      }
      return SUCCESS;
}
```

Figure 6:- Implementation of Kernel Interface.

Function encap_oa could probably be derived from the traffic selectors [rfc4306]. In the net link kernel Implementation pluto does the same as we do here but it uses encap_oa in the pfkey implementation. BUT as /usr/src/linux/net/key/af_key.c indicates that the kernel ignores it anyway. Does that mean that NAT-Traversal encapsulation doesn't work in transport mode? No. The reason the kernel ignores NAT-OA is that it recomputed (or, rather, just ignores) the checksum. If packets pass the IPsec checks it marks them "checksum ok" so OA isn't needed.

8. Conclusion

NAT is a mechanism which brought momentary abandon to the problem of shortage of IPv4 addresses. Unfortunately, it also brought some problems which we solved as described. Security is an essence part of this protocol and therefore implementation procedure is used for NAT detection. After detection of NAT box, there are appropriate actions as described. Support of NAT traversal in IKEv2 implementation solved one of the important demands for IKEv2 implementations and made this implementation more general and therefore, more appropriate to use in the IPsec. We have also shows the integration of IPSec with NAT. So for proper securing and fully functionality of NAT traversal, it should be IP Security Protected. It contains the details about patches applied and code changes done by me in Linux kernel, mipv6 Daemon and strongSwan in different releases of DSMIPv6 to implement IKEv2. The implementation of this release was to demonstrate, Movement of MN from HL to IPv4 network and Movement of MN from IPv6 to IPv4 network and vice versa.

9. References

[1]. H. Soliman, Ed., Elevate Technologies, November 3, 2008. Mobile IPv6 Support for Dual Stack Hosts and Routers draft-ietf-mext-nemo-v4traversal-06.txt.

[2]. Vida, R. and L. Costa, Eds., "Multicast Listener Discovery VeMyon 2 (MLDv2) for IPv6", RFC 3810, June 2004.

[3]. Perkins, C., RFC 3344, August 2002. "IP Mobility Support for IPv4".

[4]. Johnson, D., Perkins, C., and J. Arkko, RFC 3775, June 2004. "Mobility Support in IPv6".

[5]. H. Soliman, Ed., Elevate Technologies, November 3, 2009. Mobile IPv6 Support for Dual Stack Hosts and Routers draft-ietf-mext-nemo-v4traversal-08.txt.

[6]. NEPL (NEMO Platform for Linux) how to, June 24th, 2009.

[7]. MIPL (Mobile Ipv6 for Linux), how to, 2004-4-20.

[8]. Conta, A. and S. Deering, "Generic Packet Tunneling in IPv6 Specification", RFC 2473, December 1998.

[9]. K.L.Bansal, Chaman Singh, "Dual Stack Implementation of Mobile IPv6 Software Architecture", IJCA- Volume 25, No 9, July 2011.

[10]. Sebastien Decugis, Nautilus6," How To: Dynamic keying for Mobile IPv6 using racoon2 and mip6d". September 2007.

[11]. Yoshifuji Hideaki and al., In special section on internet technology IV, IEICE Trans Comumun, Vol.E87-B, No3 March 2004. Linux IPv6 Stack Implementation based on Serialized Data State Processing.

[12]. Arkko, J., Devarapalli, V. and F. Dupont, RFC 3776, June 2004. "Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents".

[13]. F. Audet and C. Jennings. Network Address Translation (NAT) Behavioural Requirements for Unicast UDP. RFC 4787, Internet Engineering Task Force, January 2007.

[14]. K.L.Bansal, Chaman Singh, "NAT Traversal and Detection on Dual Stack Implementation of Mobile IPv6", IJCA- Volume 29, No 7, September 2011.

[15]. Egevang, K. and P. Francis, The IP Network Address Translator (NAT), RFC 1631, May 1994.

[16]. Kent S, and K. Seo, Security Architecture for Internet Protocol, RFC 4301, December 2005.

[17]. Kent, S., IP Encapsulating Security Payload (ESP), RFC 4303, December 2005.

[18]. Kent, S., IP Authentication Header, RFC 4302, December 2005.

[19]. C. Kaufman, Ed., Internet Key Exchange Key (IKEv2) Protocol, 2005. URL: http://www.ietf.org/rfc/rfc4306.txt

[20]. Holdrege, M. and P. Srisuresh, Protocol Complications with the IP Network Address Translator, RFC 3027, January 2001.

[21]. Chaman Singh, S Kumar, S Kumar, K.L.Bansal," Design and Implementation of Mobile IPv6 Data Communication in Dual Networks", IJCSI - Volume 1, Issue 9, Page N0. 182-190, January 2012.